Premier 3

The Premier User's Guide

Circle Software

Premier 3

The Premier User's Guide

Circle Software

Premier 3.1

The Premier User's Guide.

First edition, March 1990
Second edition, August 1990.

All rights reserved. No part of this manual may be reproduced in any form whatever, without the written consent of *Circle Software*. The only exceptions are as provided by the Copyright (Photocopying) Act, for the purpose of review, or for any software listed herein to be entered into a computer for the sole use of the owner of this book.

While every care has been take in the design and content of this book, and the software to which it refers, no responsibility can be accepted by *Circle Software* for any loss of data, time or money, or other consequential effects resulting from it's use, however caused.

While the command language described remains the property of Circle Software, use of the language by third parties to create Premier applications for publication is encouraged. The sale of such applications is subject to quality approval by Circle Software.

Premier is the copyright of -

Circle Software,
33, Restrop View,
Purton, Swindon,
Wilts. SN5 9DG
Tell: 0793 770021

Contents

Preface 1
The Premier System 1
Command Files 1
About this Manual 2

Introduction. 3
Installation 3
Security and Validation. 4
Editing the !Run File 5
System Windows. 6
Loading Files. 6
The Main File Types. 7
In Memory Text Transfer. 9
Printing Documents. 9
File Saving. 10
Closing and Opening Windows. 10

Dialogue Boxes 13
Text Input Boxes. 13
Push Buttons. 14
Other Icons. 15

The Icon Bar Menu 17
Info. 17
Create. 17
Re-Open 18
Applics. 18
Command. 19

Colours. 19
Messages 20
Quit 20

The Command Window 21
Why Use It? 21
Entering Simple Commands. 22
The Command Store. 23
Star Commands. 24
The Journal Window. 24

Basic Concepts 27
Files, Buffers and Windows. 27
Key Definitions. 29
Variables and Constants. 31
System Variables. 32
The Menu Command. 33
Other Simple Commands. 33

More Complex Commands 35
Conditional Commands. 35
Logical Expressions. 35
Repetitive Commands. 36
Command Lists. 37
Macro Commands. 38
Command Formatting. 39
Functions. 39
Passing Parameters. 41
An Example Macro 43

Errors and Messages 45
Message Severity Levels. 45
Controlling Message Reporting. 46
Saving and Restoring Levels. 47
Program Bugs. 48

Editing Commands. 49
Marking Text. 49
The Cursor Marker. 50
The ClipBoard Buffer. 50
Cutting Marked Areas. 51
Copying Marked Areas. 52
The Paste Command. 52
Selecting Different Markers. 53
The Find and Search Commands. 54
Searching Backwards. 55
The Finder Dialogue Window. 55

Saving and Spooling 57
The Save Command 57
Spooling. 57
The Filer Window. 58

Expressions & Operators 59
Integer Expressions and Operators 59
Integer Conversion Operators 60
String Expressions and Operators. 61
Non-Printing Characters. 62

Dynamic Text 65
What is dynamic text? 65
How is it Done. 65
Simple Items. 66
Direct Commands. 66
Direct Expressions. 67
Direct Function Items. 67
Direct Item Compilation. 68
Entering Dynamic Text. 69
Editing Dynamic Text. 69
Font Control. 70
Using Functions. 71
Headers and Footers. 72
Document Formatting. 73
Nesting Dynamic Items. 73

Data Base Facilities. 75
What is a Data Base. 75
Compatibility. 76
Opening a Data Base File. 76
Reading Successive Records. 77
Searching for Records. 77
Using Filters. 78
Indexing. 80
Special Field Variables. 81
The Data Base Dialogue Box. 82
Setting Filters 83
The Action Buttons. 84
Entering & Editing Records. 85

Printing 87
Single Sheet Printers. 88
Hidden Text 88
Control Codes. 88
Initialising the Printer. 89
Printing Pound Signs. 89

User Defined Menus 91
Menu Definitions. 91
Displaying Menus. 93
Sub-Menus. 93
Other Uses for Menus. 93

Formatting the Date 95
Date Variables. 95
The Time and Year. 95
The Date Format String. 95
The Special Format Characters. 96

Command Files. 99
Command File Execution. 99
Creating Command Files. 101
Document Format Files. 101
Application Command Files. 102
User Defined Applications. 102
Some Useful Macros 103
A Data Entry Macro 104
A Relational DataBase Example. 105
Simple Text Processing. 106
Debugging Command Files & Macros. 107

The Config File. 107

Index 109

Preface

The Premier System

Premier is a command driven textual information processor, capable of all manner of text manipulation. Nearly everything Premier does, it does in response to commands given to it, using a powerful, yet easy to use command language. These commands may be contained in a textual command file which Premier will read and execute, or they may be typed directly into Premier's command window.

In fact, Premier does not even know what to do when the letter 'a', or other key, is pressed on the keyboard, unless it has been told! Fortunately, a command file is provided, called `StdKeyDefs`, which Premier may be told to read automatically whenever it is started.

This idea is at the heart of Premier. Using such command files, the program may be configured to perform any function, and provide any facility.

Command Files

Of course, if everything we needed to do involved typing commands at the keyboard, the program would not be very convenient to use. However, the commands provided allow quite complex actions to be defined and assigned to any key, including function and cursor keys, as well as mouse buttons. Commands also exist for creating wimp menus, and assigning actions to the menu options displayed.

It can be seen therefore, that using the command file principal, Premier may be configured to provide any facility

involving the manipulation of text, which includes most computer applications such as word processing, data base management, mail merging, etc, and that the end result will be an easy to use, professional, multi-tasking, yet custom made, desktop application.

About this Manual

The quickest way to start using Premier, is by loading any of the ready made applications available, However, if you want to do more than this, perhaps to modify an existing application or to write your own, then the content of this manual should be studied first.

In addition, since all the facilities described in this manual are fully available within any application, whether they are made specific use of or not, it follows that you will more fully understand the basic principles involved, and therefore get more out of your application, if you are familiar with the contents of this manual.

We will deal with Premier at the *command level*. That is, with the basic facilities available within Premier itself, and with the commands required to make use of them. It is arranged in the form of a work through tutorial, and should be read chapter by chapter, in the order presented.

The first chapter deals with the installation of Premier, and explains some of the actions Premier will take in response to desktop events.

The following chapters deal with the main facilities, with worked examples. Although it may be useful to read these chapters before turning to your computer, the best way of learning anything is to do it, and we recommend that you work through the examples, in the given order.

Introduction.

This chapter describes how to install and run Premier, and how to carry out some basic actions. It is assumed that you are familier with basic WIMP operations, and understand what is meant by *dragging* the pointer or mouse, *clicking*, double clicking, etc. If this is not the case, you may need to refer to the Welcome and User guides supplied with your computer.

In addition to responding to direct commands, Premier naturally responds to various events within the desktop world, such as icons being dragged onto it's icon on the icon bar, or into one of it's windows. These and other actions are described.

Installation.

First make a working copy of the disc supplied, and put the original in a safe place. If you only have a single floppy drive, first create a RAM disc of at least 250K bytes, by dragging the appropriate bar on the WIMP task display.

The Premier icon

Place the Premier disc in the floppy drive, and click on both the floppy and RAM drive icons to display their directory viewers, and drag the !Premier icon from one to the other.

Next place a new blank disc in the floppy drive, and if necessary, format it by clicking the menu button over the floppy icon, and choosing the format option. This menu also allows you to give the disc a name. Next simply drag the !Premier icon from the RAM directory viewer into the new floppy disc viewer. It is also necessary to repeat the operation with the !System application supplied, as the utilities contained in this must be available for Premier to run.

If you have two floppies, or wish to install Premier on a hard disc, you can of course drag the relevant icons directly to their new destinations. In the case of a hard disc, you may not need to copy the !System files if these are already available, but please ensure that the files you have are at least the same, or later versions of the ones supplied on the Premier disc. If this is not the case, you will get an error when you start Premier running.

Security and Validation.

Premier is protected by a security system, designed to protect our sales, without inconveniencing you with dongles, or copy protected discs.

All copies of Premier are supplied with an embedded serial number, which uniquly identifies your copy. As supplied, Premier will NOT perform any file saving or printing operations, until it has been validated. To do this proceed as follows.

First double click on the !Premier icon within the directory viewer. After a few moments, the Premier icon will appear on the icon bar. Press the menu button over this icon to display the Premier menu, and select the Info option. An information window will be displayed, showing the serial number of your copy. Full instructions for validating your copy are given on the lower half of this window (use the scroll bar).

Validation involves obtaining a Password from Circle Software, by quoting your serial number. This password should first be entered into the Info window, so that Premier may save edited files, and then, to avoid having to do this each time, the password should also be entered into the !Run file within the main !Premier application directory.

The following describes how this may be done using Premier, and provides a useful, hands on introduction.

Editing the !Run File

To do this, find the `!Run` file by opening the `!Premier` directory, (hold down shift, while double clicking on !Premier), and drag it's icon onto the !Premier icon on the icon bar. A window will be opened showing the `!Run` file contents.

Next, click on the toggle size icon (top right corner of window) to expand it, and use the scroll bars if necessary to locate the `Run` command near the bottom of the file. You will see two such command lines, the first is commented out, and is an illustration of how the second should look when you have edited it. The password (number) shown is an example only; you must obtain your own from Circle Software. To edit the real (second) Run command, click the pointer on it to position the caret, and use the cursor keys to position it at the end of the word `PASSWORD`. The **Delete** key may then be used to delete the word, and the correct password typed in it's place.

To save this file back to disc, toggle the window size back, exposing the icon bar, click the *menu* button over the !Premier icon on the icon bar, and select the `Command` option, to invoke the Premier command entry window. Then type the following command at the displayed caret;-

```
filer
```

This should be followed by the **Return** key, which will cause the Premier filer window, or dialogue box, to be displayed.

Unfortunately, the computer still has this file open from

when Premier was started, and it will not be possible to save the new version unless we change it's name. To do this, click in the `filename` window, and add an 'X' say, to the end of the name. Clicking on the `Save` button will now cause the file to be saved.

After completing this Premier session, use `Quit` from the icon bar menu, and re-name the edited `!RunX` file to `!Run`, so that when Premier is run on subsequent occasions, the password will be obtained from this new !Run file.

System Windows.

Premier always has two special windows pre-defined when started up, and these are the Journal Window, and a ClipBoard Window.

The Journal window is used to display such things as error and other messages, and to journal all commands entered in the command entry window. It may often be useful to consult this window when things don't seem to have gone according to plan!

The ClipBoard Window may be used to display whatever is currently on the clip board, which is the default destination of all *cut* or deleted text.

Both of these windows may be opened from the icon menu's `Re-Open` option, and closed in the normal way.

Loading Files.

There are several methods of loading files into Premier, and these conform to the Acorn desktop standard.

In general, files may be loaded by dragging their icons from a

directory viewer, or from a save dialogue box of another RISC OS compatible program, onto the Premier icon on the icon bar.

Files may also be loaded, depending on their type, by double clicking on their icons within a directory viewer. This will also automatically install Premier on the icon bar, if it is not already there.

The term loading a file, has been used here in a general way. What Premier actually does with a file when loaded as described, depends on it's file type, as described below.

The Main File Types.

Premier recognises the three private file types, *PremDoc*, *PremCmd*, and *PremData*, as documents, command files, and database files respectively. The different ways these are handled is fundamental to the way Premier works.

The *PremDoc* file type is the main Premier document file, and is represented by a *scroll* icon. When loaded, Premier will look for a *PremCmd* file having the same name, within the `Commands` directory, and if it finds one, will load and execute it. This file is optional, and if created would normally contain formatting and other commands required for the correct display of the document.

Premier will also load a command file having the same name, but within the `Commands.Format` sub-directory. This file is created automatically for internal use, and should not be modified by the user.

Premier Command files, have the file type *PremCmd*, and are represented by the *cut and paste* icon, symbolising text

manipulation. When clicked within a directory viewer, or dragged onto the Premier icon on the icon bar, the commands contained within them are executed, and any errors encountered are reported in the Journal window.

If all is well, a compiled version of the file will be saved within the `Object` directory, to be used next time the command file is loaded. This saves considerable time on subsequent uses of the file. The action is fully automatic, and you will normally be unaware of what is happening. However, you should be prepared for a longer loading time when a command file is first used after being modified in any way.

The third special file is the Premier *Data file*, having the *PremData* file type, and the floppy data icon symbolising disc based data storage. These types are not loaded at all, but are opened and the first record read. Many files may be opened in this way without limit, but the last to be opened is always considered the current one. Since only the current record is actually held in memory, data files larger than the available memory can be employed.

If is is required to view either a *PremCmd* or *PremData* file in a window, this may be done by first creating an empty window, and dragging the file's icon into it. This type of *load at cursor* operation is described later.

Other file types recognised are the plain text files defined by Acorn, such as *Obey*, *Command*, *Text*, *PoScript*, etc. These are all treated as *text* files by Premier, and will be loaded into windows when dragged to the icon bar, or into a Premier window. Premier will also load these types when they are double clicked within a directory viewer, provided some other installed disktop program does not get there first!

In Memory Text Transfer.

Consistent with the Acorn WIMP standard, Text may be transfered to and from other multi-tasking programs, by dragging icons between them.

For example, a text file may be transfered from *!Edit*, by choosing the `Save` option from the Edit menu, and dragging the file icon, either onto the Premier icon on the icon bar, or directly into an open Premier window. When this is done, the data is copied directly between the two programs, without involving disc operations. This action is dependent on the other program being equiped with this facility, and if this is not the case, transfer via a temporary disc file will be accomplished automatically.

Printing Documents.

Correct printing of a Premier document which contains special fonts, etc, is dependent on Premier *knowing* the right code sequences needed to accomplish these effects on your printer. These codes are defined in a series of Premier *command files*, one of which is executed automatically when Premier is invoked.

Several such files are provided to suit different printers, and if one has not been executed, it is only necessary to double click on the appropriate one, to ensure correct printing. These files are fully described in a later chapter.

Assuming the correct printer code file has been executed, there are two ways a file may be printed. The simplest is to click on the open window to select it (indicated by the title bar becoming a lighter colour), and typing the **Print** key. This is to the right of function key F12.

An alternative, assuming a suitable printer driver is installed on the icon bar, is to open the *Filer* dialogue box by issuing the `filer` command, and drag the file icon onto that of the printer driver.

Note that this method, after a suitable conversation between Premier and your printer driver, still results in Premier sending the text to the printer in the same way as when the **Print** key is used, and the same remarks concerning control codes applies. Premier does not use the *graphic dump* method as required when using anti-aliased fonts, etc, which would not be efficient in this context.

One advantage of this method is that the text (with control codes) is first spooled to a temporary disc file, from where the printer driver will process it in multi-tasking mode, allowing continued use of Premier while printing is taking place (assuming the printer driver can operate in this way).

File Saving.

Consistent with other desktop applications, Premier files may be saved by dragging icons into the appropriate directory viewer, and in this case the file name used will be the last element, the leaf name, taken from any existing file name. A suitable icon is displayed in the `filer` window, as already seen.

Direct `save` commands may also be given in the command window, and the use of this is also described fully in the appropriate chapter.

Closing and Opening Windows.

Premier windows may be closed by clicking on the close icon at the top left of the window in the normal manner. However,

the data displayed within a closed window is not lost, and therefore no *file save* prompt is given.

Any closed window may be instantly re-opened, by choosing the ReOpen option from the icon bar menu. This will reveal a sub-menu containing all windows currently in existence. Closed windows may be reopened simply by selecting them from this sub-menu.

This system allows you to keep the desktop tidy by removing unwanted windows, without the necessity of saving and reloading later.

The Re-Open option will also show other system windows, such as the *ClipBoard* and *Journal* windows, while modified windows are flagged with an asterisk, as in the window title bar.

It may be noted here, that Premier windows are always stacked in the top right hand corner of the screen when first opened. From here they may be toggled to full size and back without loosing their position in the stack, provided they are not moved by dragging their title bars, or re-positioned in level using their *back* icons. This is a convenient way to keep the desktop tidy, and all windows immediately accessible.

Dialogue Boxes

This chapter describes the features to be found within the various dialogue boxes provided within Premier, and their method of use. For a more detailed description of individual dialogue boxes, you should refer to the corresponding chapter.

A Premier command is provided to display each box, and these commands will normally be assigned to a function key, for immediate access within any specific Premier application. The appropriate user guide and key strip should be consulted.

Text Input Boxes.

When an action requires you to enter data, an input box will be provided. This will be a white box containing the text caret, ready for you to type into. If the caret is not displayed in the required box, clicking the select button while the pointer is over the box will cause the caret to appear at the pointer position within any displayed text.

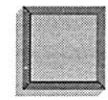
Within these input boxes, full editing of the entered text is possible, using the cursor keys, and the delete and copy keys, as required. The full range of editing actions available is fully compatible with other RISC OS applications, and is summarised below.

If the text entered is longer than the box, the display will scroll sideways as required. The program will refuse to respond to key strokes if the maximum text length has been reached, or because a particular key is invalid in that context. In most cases, use of the return key is required to terminate entry, and cause the displayed text to be used by the program.

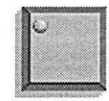
Keys.	**Action.**
Delete	Delete character to left of caret
Copy	Delete character to right of caret
Shift + Copy	Delete to end of word.
Control + Copy	Delete to end of line.
Cursor Left	Move caret one place left
Cursor Right	Move caret one place right
Shift + Cursor Left	Move caret one word left
Shift + Cursor Right	Move the caret one word right
Ctrl + Cursor Left	Move left to end of line
Ctrl + Cursor Right	Move right to start of line

Push Buttons.

There are mainly two types of buttons provided within dialogue boxes, those which select an option of some kind, and those which initiate an action. Both types represent push buttons, and will appear to be either in or out.



Action buttons will 'depress' themselves automatically whenever the pointer is over them. To initiate the indicated action, simply click 'select' while the button is shown depressed.



Option buttons additionally have an 'LED' lamp, illuminated when the button is 'depressed', indicating that the appropriate option is currently selected. To select or deselect the indicated option, simply click 'select' over the button to toggle it's state.

Some options may be non-selectable, depending on the state of other option buttons, where applicable.

Other Icons.

Various other icons may be shown within dialogue boxes. These are normally coloured orange, and will change to black when the pointer is over them. A typical example is the arrow icon, used to step through a list of values or other items, the current value being shown in an adjacent white box.

Other specialised icons are described within the description of their dialogue boxes.

The Icon Bar Menu

This chapter describes the use of the menu available from the Premier icon on the Icon bar, and describes the options available. Some of these options are described more fully in later chapters.

To display this menu, click the menu (middle) mouse button, while the pointer is over the Premier icon, on the icon bar.

Info.

The first option, provided as standard in most RISC OS applications, displays a window containing information about the program. However, in Premier's case, this contains more than usual.

The main addition is the serial number displayed near the bottom, which is unique to your copy of the program, and provided for security purposes.

This window is also scrollable, the lower half displaying instructions for validating your copy, and a password entry window, as explained above.

Create.

This option allows the creation of new documents, and a sub-menu offers a list of document types, covering most of the text file types defined by Acorn, as well as the types allocated to **Circle Software** for exclusive use within Premier.

Selecting a document type will create a window called

Window. which will be displayed on the screen. When a second window is created, this will be called *Window-1*, etc.

Note that a short cut allows you to create a new `PremDoc` window, simply by clicking *select* over the Premier icon,

Windows having any desired name may be created using the WINDOW command, described in the Reference manual.

Re-Open

When you close a Premier window, data is not lost, and it is not necessary to save the document to a disc first. The `Re-Open` option leads to a sub-menu containing a list of all known Premier windows, from which any closed windows may be re-opened. Windows aleady open and shown faded, and are not selectable.

Note that this sub-menu also allows the pre-defined **Journal**, and **ClipBoard** windows to be opened. These are described later.

Applics.

This option displays a sub-menu containing a list of all command files within the `Commands.Applics` directory. This directory should be reserved for application command files, either ready purchased, or your own.

When any of these files is selected, the file is executed, to configure Premier for the selected application, and this will usually cause a master menu for the application to be defined. Where two or more applications are loaded, a top level *Premier* menu will be created, giving separate access to the individual application menus.

Command.

This causes the command entry window to be displayed, as mentioned above. The use of this is fully described in the next chapter.

Colours.

Premier displays various items within it's windows in different colours, so that they may be easily identified. These items will be displayed on a sub-menu, and include `Paper`, `Markers`, `Margins`, and `Controls`. To change the colour of any of these, select the colour sub-menu from the option required, and click on the new colour.

The `Markers` option refers to the colour of marked areas, while `Controls` refers to the colour of control characters, such as tabs, newlines, etc. `Margins`, refers to the margin outline box, and the shading used in the perforation skip area, if any. To make these items invisible, simply select the same colour as the paper. The position of dynamic items may be hi-lighted by choosing a colour for the `Dynamic` option.

None of these options affect printed output, and are only provided to enable you to make a personal choice, or to make an item invisible. All colours may be pre-set by including the `COLOURS` command in the `config` file, described in a separate chapter.

The `Dynamics` option allows a colour to be selected to indicate the position of dynamic items within the text.

The *paper*, or *background* colour will only apply to windows created *after* this option has been used, allowing windows with different colour backgrounds to be created if required.

Messages.

This option allows user control over the type of message required to be reported, as shown in the sub-menu. Briefly, messages are assigned a level of importance, from VERBOSE, which reports just about everything that happens, to FATAL, which means Premier cannot continue reliably. By default, all messages of ERROR or worse are reported. See the chapter Messages, for more details.

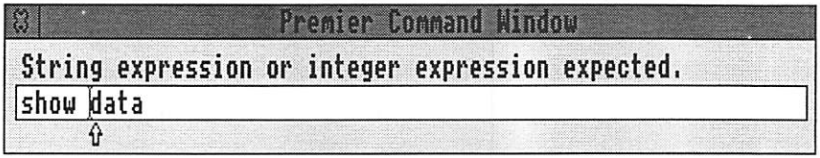
Quit.

This allows the current Premier session to be terminated, and removes the Premier icon from the icon bar. If there are modified, unsaved documents (*excluding* the Journal and ClipBoard), this fact will be reported to give you a chance to save anything required before the program terminates.

The Command Window

This chapter describes how to use the *command window* to issue direct commands to Premier, and should be read before going on to the following chapters.

To display the command window select the `Command` option from the icon bar menu.



Why Use It?

If you are familiar with other RISC OS applications, you may be wondering why you need a command window, or indeed why you should need to use typed commands at all. We hope that the next few chapters will answer these questions more fully, but a brief explanation may be in order here.

Most conventional programs attempt to offer all the facilities the authors think you may reasonably require, and which they can provide within the time scales, and financial constraints imposed by the need to produce the software in good time, at an affordable price. In the main, they do a good job, but it follows that in practice there are many facilities which *you* require, but which have been left out.

In designing Premier, we have recognised from the beginning that if the software is to meet *your* needs, it must have inherent flexibility. It must put *you* in control, and not expect you to live within the constraints imposed by the program.

The main strength of Premier is therefore it's command

language, which allows *you* to configure just about everything to suit the way *you* work, and the precice job *you* have to do. The price of this flexibility is the need to learn how to tell Premier what you want it to do, and this is the subject of the next few chapters. You will find however, that this has been made as simple as possible.

You will also find that a great deal may be achieved through the use of dialogue boxes and menus, without using a single direct command, or that the facilities provided within a specific Premier application are sufficient to meet you immediate needs. However, you will find, sooner or later, that a knowledge of Premier's command language will enable you get more from the program than would otherwise have been possible.

Entering Simple Commands.

Select the `Command` option from the icon bar menu, and type the following command, followed by the Return key.

```
show data
```

This command has an error, and this will be reported above the text entry box, while an arrow below it will indicate the position where the error was detected. Also, the caret will be positioned before the offending character, ready for you to correct the mistake, by editing as described in a previous chapter.

The `show` command may be used to display a variety of things, but the word `data` is clearly not one of them. What we intended to type was `date`. To correct this, type **Ctrl + CursorRight**, to position the caret at the end of the command, delete the 'a' with the **Delete** key, and type 'e' in it's place. You could just as easily have typed **Ctrl + Copy** to

delete the word `data`, and retyped `date`. Now type **Return** to re-execute the command.

If all is now well, Premier will display today's date in a message window. This window, and the command window itself, may be removed when no longer required, by clicking on their *close* icons in the normal way.

To show what else Premier can do, try the following, exactly as shown.

```
show "I am", 3*(4+5)-6, "years old."
```

This should display the correct (!) answer in the message window. You may like to try showing `time`, `stddate`, `year`, `dateformat`, or `free`, using the show command. All of these items are explained in the *Premier Reference Manual.*

The Command Store.

Premier incorporates a useful feature, whereby the last 20 odd *successful* commands are stored, and may be instantly recalled for re-use.

To recall that last command executed, simpley type the **down cursor** key. Assuming several commands exist in the store, further presses of this key will reveal earlier commands, in reverse order. The command list will be wrapped round back to the start, when the end of the list has been reached.

If the **cursor up** key is used, the command list may be stepped through in the opposite direction, allowing any previous command to be selected.

When any command is displayed within the window, it may

be further edited if required, and then re-executed by typing the **Return** key.

Star Commands.

This may be an appropriate moment to mention Premier's ability to execute system *star* commands, directly from the command window.

The asterisk '*' symbol is recognised by Premier as a command word, and may be followed by any valid string expression. This may be a string variable name (see later), text within double quotes, or a combination of these.

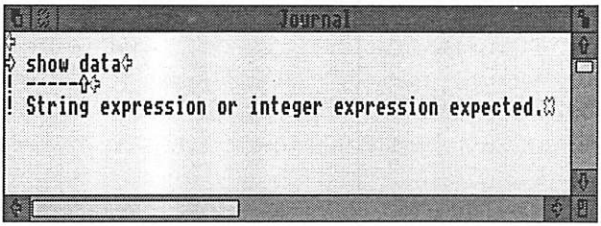
Any command so given, is sent to the computer operating system without being checked, and is equivalent to typing commands in the `'New Task'` option from the task handler icon (the 'A' icon on the bar), or typed after using function key F12.

One advantage of this facility, is the ability to include system commands within a Premier command file, to configure the desktop, set default directories, or install other desktop tasks, perhaps when Premier is started up.

The Journal Window.

All commands entered via the command window, are copied into a special *journal* window, together with any error or other messages which may result. To display this window, select the `Re-Open` option from the icon bar menu, and select the `Journal` option from the sub-menu of known windows.

The Journal window also contains reports of errors found while executing command files, and provides a useful



reference to past events. The use of command files is the subject of another chapter.

It should be noted that while the Journal window is open, certain errors and messages will *only* be reported in this window, and will not be duplicated elsewhere.

The Journal window is just like any other Premier window, and may be edited or saved, etc, as required. Messages will always be added at the end, regardless of where the cursor happens to be, should this have been moved.

Basic Concepts

This chapter explains some of the basic Premier concepts, and provides an introduction to the Premier command language. The examples given should be followed, by typing them into the command window, exactly as they appear.

Files, Buffers and Windows.

Premier is concerned with *information*. Information is facts and figures, and these are usually represented by words, or text. This text may exist on a floppy disc in the form of a text file, or we may wish to enter new information from the keyboard, which we wish to be stored in a text file.

Premier is capable of prompting us to enter information, and writing this directly to a file. More usually, our new information, or that read from an existing file, will need to be held in the computer's memory, while we perform some operation on it, or simply display it on the screen.

We therefore need to start by telling Premier to set aside an area of memory to hold our information. Most likely, we will want several sets of information to be held separately, and to be able refer to these independently. The memory areas used by Premier are called *buffers*, since they act as a buffer for the data, between ourselves, and the disc on which it will eventually be stored. These buffers may be refered to by their *buffer names*.

Our first Premier command is therefore the following:-

```
buffer "fred"
```

If you type this into the command window, Premier will create a buffer called *fred*, into which we can enter text ready

for processing. One point worth noting at this stage, is that the buffer name was given between quotes. Such items are referred to as *literal strings*, since they represent *literally* the string of letters between the quotes. There are other ways of representing names, and we will come to these later.

Next we need to get some text into this buffer, and this can be done in many ways. First however, since we may have several buffers, we need to tell Premier which one we are currently working on, and this is known as the *currently selected buffer*. Our second command is therefore:-

```
select "fred"
```

Premier now knows which buffer we wish to use, and will apply all the following buffer commands to `fred`. We can now insert some text:-

```
insert "This is some text."
```

The *literal string* given will be inserted into `fred`. If we do this again, Premier will insert the new text at the end of the last lot, that is, after the full stop, or at the position last reached. But what if we want to insert at the beginning again, that is, before `This` ?

Premier always keeps a note of where it has got to within a buffer, and this is known at the *cursor position*. There are, as you might have guessed, commands for moving the cursor position left, right, or even up and down within a page of text. Thus if we can give the commands:-

```
cursor left
insert " in the buffer fred"
```

These will first move the cursor one place left, then insert the

new text string between `text` and the full stop, to form a new sentence.

So far, all these commands have been issued *blind*, without you being able to see what you have done. This type of processing is often carried out from within a command file, perhaps to carry out a sequence of global edits to several text files.

To see what we have done, we need to create a window, and display the buffer in it. This actually requires three commands:-

```
window "bill"
assign "fred", "bill"
open "bill"
```

You have now created a window called `bill`, assigned the buffer `fred` to it, and lastly opened the new window, which should be visible on the screen, showing the text entered previously. If all this has seemed simple and obvious, then that is how it should be. Most of Premier's commands are simple to use and easy to remember, although they perform some quite complicated operations.

It may also seem like a long winded procedure, just to open a new window! Remember however, that we are specifically introducing you to the command language, and that you would normally use these or similar commands within a command file, function key or menu definition, and not type them directly as we have been doing.

Key Definitions.

We could carry on as before entering text using the `insert` command in the command window, but this would obviously

be tedious. All we need to do is click on the displayed window to select it, then start typing. But what actually happens when we do this?

Well, the mouse buttons and the keyboard keys were assigned actions in a command file which was executed when you started Premier running. For example:-

```
key 65  = insert "A"
key 396 = cursor left
key 256 = edit mousewindow
```

The first of these commands defines the key labelled `A` to insert the letter `A` into the currently selected buffer. All the keyboard keys are assigned in this way. The second assigns the `cursor` command which we met above, to the **cursor left** key, while the last assigns the `edit` command to the mouse *select* button. The function keys are similarly assigned suitable actions within the various Premier application command files. All keys are known by numbers, and these are given in the *Premier Reference Manual.*

The `edit` command, followed by a window name, causes the named window to be opened and given the *input focus*, ready for editing. The `mousewindow` used here is a system variable (see later) having a value equal to the name of the window over which the last button click took place. Since the select button is key number 256, the whole command causes the window clicked over to gain the input focus, and be hi-lighted.

You should now have some insight into how Premier works, and how applications may be built to provide almost any text manipulation program we might require.

Variables and Constants.

We have used the word `left` in the `cursor` command, and `mousewindow` in the `edit` command above, and these were not given within quotes, as was done, for instance, in the `buffer` and `window` commands.

When words or phrases are entered within quotes, they are called *literal strings*, or *string constants*, because we intend them to be taken literally, and they are *constant*, and cannot change their values. The items `left` and `mousewindow` on the other hand, may change values, and are therefore known as *variables*. The first is an *integer variable*, as it actually represents a number, the second a *string variable*, as it represents a name.

The variable `left`, actually represents a number which has been defined within Premier to mean *move left*. This value will in fact never change, although in theory it could, perhaps in a future version of Premier, but in any event, it will always mean the same thing.

The variable `mousewindow` on the other hand is a true variable, a string this time, and will always represent the name of the window over which the mouse pointer was last *clicked*.

The `edit` command given above therefore actually says, `edit "bill"`, if we clicked over window bill, but would say `edit "other"`, if we clicked over a window of this name. In this way, by using variables, we may make the same command actually mean different things, according to the context at the time we use it.

As you might expect, you may create your own variables, both string and integer (numeric), and use them whenever a

string or integer is required in a command. For example:-

```
str "mybuffer" = "fred"
```

Note that we must use *string constants* within quotes, for the reasons explained above. However, we can now use the string variable `mybuffer` in a command or key definition, where the actual buffer referred to may change. For example:-

```
key 384 = print mybuffer
```

This would always print whichever buffer happened to be represented by `mybuffer` at the time we typed the key. To change the value of `mybuffer` we need only use the command shown below. The `str` prefix is now not needed, as Premier knows that `mybuffer` is a string.

```
mybuffer = "anotherone"
```

Typing key 384, (the **Print** key), will now cause a different buffer to be printed out, without having to redefine the key.

It should now be clear how we may use fixed key or menu definitions etc, to refer to different buffer or windows, using variables. If you have problems with this, re-read the above now, and experiment some more.

System Variables.

The pre-defined variable `mousewindow` used above, is just one example of a range of such variables, known as *system variables* because they are defined within Premier, and their values are maintained by the program. In fact it is not possible for you to assign values to these items yourself.

Other system variables provide such values as the current

cursor line and column positions, the currently selected buffer and window names, as well as the current date and time, etc. A full list is given in the *Premier Reference Manual.*

The Menu Command.

To get a feel for the true power of the Premier command language, type the following commands;-

```
menu "Numbers" = "One,Two,Three"
offer "Numbers"
```

This will create a menu and display it on the screen. If you select an option from this menu, you will get an error message. If the menu contents had consisted of macro names, the macro selected would have been executed. We will explain this in the next chapter.

Other Simple Commands.

We have so far introduced only a few simple Premier commands, but you should already have a feeling for how commands work, and how they are put together. The *Premier Reference Manual* gives the exact syntax of each command, showing how many of them may include optional items, which modify exactly what they do. You may like to browse through this manual at your leisure, and familiarise yourself with some of the facilities available.

You can only use the commands you know exist, and there are too many to describe in detail here.

More Complex Commands

This chapter describes the more complex of the Premier commands, and may be skipped on the first reading.

Conditional Commands.

There are times when we require to execute a command only if some condition is met. For instance, we may need delete a word, if it was the one we were looking for, and not otherwise. To provide for this Premier includes the `if` command, which takes the general form:-

```
if (condition) command-1 else command-2
```

When this is executed, Premier first looks at the condition in brackets, and if whatever is there is *true*, then it executes `command-1`, otherwise it executes `command-2`. Actually, the `else` part is optional and need not be present, in which case if the condition is *false*, nothing is done.

What is *true* and *false* in this context? Well if `condition` had been an integer variable, a value of zero would be taken as *false* and any other value as *true*. If we want the action to be performed if a variable is some other value, we must use what are known as *comparison operators*. For example:-

```
if ( variable > 6 ) show "Variable is large"
```

Logical Expressions.

This would cause the given message to be displayed if `variable` was greater than 6.

The *condition* in this case is an example of a *logical expression*, i.e. a statement which evaluates to a logical

result, either *true* or *false*.

There are several operators available for use within logical expressions, and these are usable between integer or string values, but both 'sides' of the expression must be of the same type.

'==' Equal to
'<>' Not equal to
'>' Greater than
'<' Less than
'>=' Greater than or equal to
'<=' Less than or equal to

Other logical operators include `and`, `or`, & `not`, which may be used to combine pairs of the above expressions, into more complex ones, as in the following expressions.

```
line > 10  and  line < 20
fred == "hello" or buffer <> mousebuffer
not found
```

The last example uses the NOT operator to *negate* the value of `found`. That is, the expression is *true* if `found` is *not true*.

Repetitive Commands.

Sometimes we may need to repeat a command or group of commands several times, until some objective is achieved, and Premier provides for this also. The `repeat` command has the following general form:-

```
repeat command-1 until condition
```

This functions exactly as you would expect, executing

command-1 over and over until the condition is *true*. Obviously, the condition expression must change as a result of command-1 being executed, or the thing will go on for ever. An example of this command in action, might be as below.

```
repeat delete until deletedchar == " "
```

This repeatedly deletes the character to the left of the cursor position, until the character deleted is a space. The variable deletedchar is another example of a *system variable*, which is assigned the value of the last character deleted. Thus this command could be used to delete a whole word, up to the last space. Note however that it would loop for ever if it reached the start of the file, and there was nothing to delete!

Another *looping* command is the while command. This behaves in a similar way to repeat, except that the condition is looked at *first*. This means that if the condition is *false* when first evaluated, nothing is done. This is in contrast the the above repeat command, which will *always* be executed once, since the condition is evaluated at the end of the loop.

```
while ( weekday == "Monday" ) show "I'm bored"
```

This will keep repeating the message all day Monday, but do nothing if executed any other day! (If you try this on a Monday, you will need the **esacpe** key to stop it, or have a long wait!

Command Lists.

In the above examples, the actual commands executed within the if or repeat commands, may actually be a list of several commands, rather than the single commands shown

for clarity. Such command lists must be grouped within curly brackets, as below:-

```
repeat { find "fred" replace "bill" }
until not found
```

This almost English command repeatedly searches for the word `fred`, and replaces each occurrence with the word `bill`, until it cannot find any more, as indicated by `found` becoming *false*. The variable `found` is a system variable, assigned the value *true* or *false* according to the result of the `find` command. The `replace` command has no action if the last string found is not at the current cursor position.

Macro Commands.

A *macro* is simply a *command list* given a name, so that the name may be used whenever the list of commands is required to be executed. In the above example, we could have used the commands:-

```
macro "swap"
{
     find "fred"
     replace "bill"
}
```

And then used:-

```
repeat swap until not found
```

This would be more useful if the two strings "`fred`" and "`bill`" were replaced by string variables, so that the same macro could be used to swap other words, without modification.

Command Formatting.

The above macro definition could have been entered on one long line, and would have to be if entered directly into the command window. However, when entering such commands in a command file, it is good practice to lay out the commands as shown, so that the structure of the command may be clearly seen.

This may not seem too important in the above case, but where the macro may include other command lists, each with it's own brackets, it can soon become unclear which brackets belong to which command list, and which list belongs to which command word. Our convention involves increased tabing of nested command lists, and this will be used in all future complex examples. Spaces are only needed to avoid ambiguity, but aid clarity.

Get into good habits now!

Functions.

These are similar to macros in construction, but represent values, and may be used just like the simple variables we have met so far.

The value represented is defined following an '=' sign, after the command list, as in the following example:-

```
int function "test"
{
     a = b * c
     result = a + 3
} = result
```

This somewhat contrived example, defines a function called

test, which computes the integer variable *result*, from various constants and variables. The *value* of the function follows the final '=' sign, and is the integer variable *result*.

Note that this is an *integer* function, as defined by the int prefix, and must have an *integer value*. We may also define *string* functions by using the str prefix, in which case the final value must be a *string expression*.

Try entering the following definition in the command window.

```
str function "TIM" = "The time is " + time
```

Here we have left out the command list which is in fact optional within functions, and have used a string expression as a final value. This consists of a string constant or literal, concatenated (by the + operator) with the system variable time.

You may test this function by entering the command;-

```
show TIM
```

To demonstarte an important difference between simple variables and functions, now enter the following;-

```
str "TM" = "The time is " + time
```

This looks the same as the function TIM, but if we now enter;-

```
show TIM, TM
```

you will find that while the first item, the *function*, gives the current time, while the second item, the *variable*, always shows the time at which you defined it. Try recalling the

command (using the **down cursor** key) and re-executing it a few times, to satisfy youself that this is the case.

Functions are always executed whenever they are used, the value of a variable is fixed at the time it is assigned a value.

Passing Parameters.

In the above example, the function `TIM` returned variable results because we used the system function time within it's definition, while the integer function `test` will always give the same result, unless `a`, `b`, or `c`, happen to change. There is another way in which functions, and indeed macros, may be made to give different results, or do different things, and this is by *passing* them values to work on.

These values are called *parameters*, and are passed to the function or macro by enclosing them within parenthesis, following the name, whenever they are used. For example, to pass numbers to our `test` function, we write-

```
show test( 6, 9 )
```

However, we now need to redefine our function, so that it may make use of these *passed parameters*,. This is done within Premier in a novel way, which is simple and flexible to use. The following illustrates the method:-

```
str function "test"
{
     a = IP(1) + IP(2)
     result = a + 3
} = result
```

Here we have replaced the variables `b` and `c`, with the system function `IP`, each time with a different parameter of

it's own, i.e. 1 and 2. This new function gives the value of any parameter passed to `test`. `IP(1)` giving the first one supplied, `IP(2)` the second, etc.

In the example above, we supplied the numbers 6 and 9 as parameters, so that `IP(1)` will have the value 6, while `IP(2)` will give the value 9. The value of `a` will therefore be 15, while that of `result`, and therefore that of the function `test` itself, will be 18, (15+3). In other words, the function `test`, always gives the sum of two supplied numbers, plus 3.

The letters `'IP'` used above, stand for Integer Parameter. We could also use `SP` for String Parameter. It makes no difference in what form the parameters are passed to the function, as `IP` and `SP` will convert them appropriately.

For instance, if we had passed `(123,time)` as parameters to a function or macro, then `SP(1)` would give the string value `"123"`, while `IP(2)` might yield the integer `19`, although the supplied parameter types are integer and string respectively, i.e. in the reverse order.

The string variable `time` would yield the value `19`, given that the time was `19:30:59`, since IP would convert as much of the string to integer as it could, and this process would stop at the first colon. It there were no leading digits, then zero would be returned.

Similarly, if a parameter is refered to which is not supplied, i.e. `IP(3)` is used when no third parameter is passed, the value of `zero`, or the null string `""` will be used, and no error reported. Functions and macros may therefore be used with variable numbers of parameters, of either type.

An Example Macro

The following is an example of a macro using one string parameter. When used, it will assume the parameter is a string giving the full path name of a file, which it will display in a new window, ready for editing.

```
macro "DisplayFile"
{
    str "buffername" = extract(SP(1), 0, "." )
    str "windowname" = buffername + "_window"
    buffer buffername
    window windowname
    select buffername
    load SP(1)
    assign buffername, windowname
    edit windowname
}
```

This could be used by typing:-

```
DisplayFile("<Premier$Dir>.Documents.Letter")
```

First, the string variable buffername is defined using the system function `extract`. This extracts the last sub-string separated by full stops, from SP(1), the passed filename. This yields the string "Letter".

The second statement creates the variable `windowname`, by adding "_window" to `buffername`, to yield "Letter_window".

Then a buffer and window having appropriate names are created using the commands we have already met, and the `load` command used to load the required file into the selected buffer. Finally, the buffer is assigned to the window,

and the window is opened for editing.

You may care to look these commands and functions up in the *Premier Reference Manual*, for more detailed information.

Errors and Messages

This chapter describes Premier's message reporting system, and explains how you may control which messages are reported.

Message Severity Levels.

Premier has a message system, which is used for all program messages, whether they are errors or just general information. Each message carries a *message severity level*, which ranges from *verbose* to *program bug*.

verbose	User feedback on everything that happens.
info	General confirmation messages.
warning	Warnings that something unexpected has occurred, or the requested action was not possible.
error	Usually the result of a command syntax error, the requested action not being attempted.
serious	A serious error, where the entered command has the correct syntax, but the compiled code could not be executed.
fatal	A fatal error is one where it should not be possible for Premier to continue, although it may attempt to!
bug	This indicates a condition which should not occur, and is not the user's fault.

All message above *info* show their severity level at the start of

the message. For example -

```
Error: "=" expected in line 27
```

Controlling Message Reporting.

It is possible to select the minimum severity level which you want Premier to report. By default, you will get messages only at or above *error* level, that it, you will not receive any messages classed as *verbose*, or *info*.

While getting to know Premier, it may be useful to have verbose messages reported, to keep you informed about what is going on. As you gain more confidence you will probably find these messages irritating. Another use for this facility is to suppress all messages below the *serious* level during command file processing.

When a command file is executed for the first time, the textual commands are checked, and if all is well, a compiled code version of the file is produced, (see later chapter). During this process, it is important that syntax error messages are reported, so Premier automatically selects *warning* level reporting, prior to executing the file. This may be overridden by placing the `messages` command as the first line in the file. When the compiled code is executed on subsequent occasions. Premier will not alter the message level.

The `messages` command selects the reporting level which you require, and prevents all messages having a lower level from being displayed. For example.

```
messages verbose
```

Here, `verbose` is a system integer constant defined to have

the required value expected by this command. There is one corresponding to each message level listed above, (See the Premier Reference Manual).

It is also possible to set the required level from the icon bar menu, by selecting the messages option, and choosing the required level from the sub-menu, as explained in an earlier chapter.

Saving and Restoring Levels.

It is often necessary to change the message reporting level within a command file, to suppress unwanted messages, say during program start up, or to turn verbose on for diagnostic purposes. When this is done, it is important to restore the previous level at the end of the file.

The system variable `messages` has been provided, which has the value of the currently set level. A command file might therefore contain the following sequence.

```
int "level" = messages
messages = serious
<Main file content.>
messages = level
dispose "level"
```

This sequence defines an integer variable `level` to have the value of the currently set message level, then sets the current level to be *serious*, thus rejecting all *verbose*, *info*, *warning* or *error* messages. At the end of the file, the original level is restored with the `messages` command specifying the original level which was saved in the variable `level`. This variable is finally disposed of, although this last action is optional.

Program Bugs.

Although unlikely, if you should happen to get a message of this type, we would like to know about it! Completely bug free programs, especially complex ones, are rare, and errors can only be corrected if they are reported.

If possible, supply us with a copy, or printout, of the journal buffer, the data or command file you were using, and a complete description of what you were doing at the time. If Premier terminates itself, as it will if it can no longer continue sensibly, a file `!Premier.ErrorLog` may be produced. This may contain information vital for tracking down the problem, and a copy of this would be most useful.

All material will be treated in the strictest confidence, and would greatly assist us in improving the program. A new program disc will be supplied free when the fault has been corrected.

Editing Commands.

This chapter describes the Premier commands associated with editing functions, and describes the use of the Find & Replace dialogue box.

Marking Text.

There two commands concerned with placing markers within a text buffer, which are used for different purposes.

During interactive editing, the `mark` command, usually assigned to a function key or mouse button, will place a marker in the text at the current caret position. When two such markers are inserted, the text between them will be shown in reversed characters, in the currently set marker colour (red by default). The marked area may be adjusted by moving the caret and again using the `mark` command.

This time, since there are already two markers placed, the last placed marker will be moved, either shortening or extending the marked area. To remove the marked area completely the `unmark` command may be given.

Markers placed in this way have the names "MarkedStart", and "MarkedEnd", and may be refered to in a similar manner to those described below.

The second command takes the following form:-

```
marker "mark1"
```

This creates a marker, again at the current caret position, but gives it a name by which it may be referenced in other commands. There is no limit to the number of markers created in this way, but they must of course all have unique

names. Such markers are generally used for manipulating text within command files, and are not shown on the screen.

Markers may be used with the *cut* and *move* commands explained below.

The Cursor Marker.

Every buffer created has it's own marker, which is created automatically, and which has the same name as the buffer. It is this marker which is displayed as the caret, when the buffer is displayed in a window. The system variable `marker`, always has as it's value the name of the currently selected marker, and may be used to refer to it, in the same way as other named markers.

The ClipBoard Buffer.

This is a text buffer created within Premier, which is used to deposit text cut from the working buffer, so that it may be re-inserted if required.

It should be noted that although this facility may be used as a holding buffer for text being moved from one position to another, it's primary purpose is to act as a safety device. If any text is cut out accidently, or you should change your mind after doing it, you may always retrieve it from this buffer. Other uses may may occur to you.

However, direct movements of text between buffers is possible without using the ClipBoard buffer.

The ClipBoard buffer may be displayed by selecting `Re-Open` from the icon bar menu.

Cutting Marked Areas.

This, as might be expected, is done using the `cut` command, which may take from zero to three marker names. If given without any markers, the currently marked text area (defined using the mark command as above) will be cut out, and copied to the cursor position within the ClipBoard buffer. If there is no marked area, an error will be reported.

One named marker may be given in the command, as below:-

```
cut "mark1"
```

In this case it will again be assumed that a marked area exists, as above, but the text marked will be cut out, and moved to the position of the *named marker*, instead of the ClipBoard buffer. This named marker may be anywhere, except within the marked area!

The third form of the command takes two marker names:-

```
cut "mark1" "mark2"
```

This time the two markers supplied will be used to define the *text area* to be cut out, and this will be moved to the ClipBoard again. The only restriction this time is that both markers must be in the same buffer!

Lastly, we may give the command:-

```
cut "mark1"  "mark2"  "mark3"
```

If all three markers are given, the area defined by the *first two*, will be cut out and inserted at the *third marker* position. This may be in the same, or a different buffer.

Copying Marked Areas.

This is done using the `copy` command, which also has the options described for the `cut` command above. The action is exactly the same, except that the original marked area is *not* deleted.

This allows text to be duplicated within the same, or another buffer, or within the ClipBoard buffer.

One use for this is to collect various areas of text from perhaps several different documents, to create a new document containing the selected paragraphs. Text accumulated in the ClipBoard buffer will remain until explicitly deleted, using the `clear` command.

The marker names may of course be provided in these commands as string variables, which may include the current cursor marker, as in:-

```
copy marker
```

This will copy the *marked area* to the current *caret position.*

The Paste Command.

This simplifies the action of copying the content of the ClipBoard buffer into a document, and takes the form:-

```
paste "markername"
```

The entire content of the ClipBoard buffer will be copied to the named marker position. To paste to the current caret position, use the marker system variable, as in:-

```
paste marker
```

Selecting Different Markers.

The `select` command introduced earlier, actually expects a *marker* name to be provided, not a *buffer* name, and we may therefore select any named marker. It is possible to create a marker at a specified position, move the cursor to another position, perhaps using the `find` command (see below), then move back by selecting the new marker.

If you want to move the *caret* between different marker positions, this too is possible using the `goto` command. For example.

```
marker "place"
find "Some text"
goto "place"
```

This sequence leaves the original cursor marker, having the same name as the buffer, at the start of "Some text", assuming this was found, and selects the marker "place" as the new caret position. This marker has now become the new window cursor, and will be selected as the caret position, whenever the window is clicked. Several such markers could be set up, and moved between as required.

The `show marker` command may be used to find which marker is currently selected. To return to the original cursor marker, the system variable `buffer` may be used, which has the name of the *current buffer*, and hence also the original marker name. i.e:-

```
goto buffer
```

The Find and Search Commands.

These are normally used within function key definitions, or in command files, as the actions they carry out are available from the **Finder** dialogue box described below.

Both commands act on the currently selected marker, and may be used to locate specific items of text. The only difference between them is that the `search` command will recognise certain characters as pattern matching characters, while the `find` command will take the string given literally.

```
find "Bone."
search "Bone."
```

The first command will locate the string given, *exactly* as it is given, if it exists. The second will use the dot as a wild card, matching *any* character. It may therefore locate "Bones" for example. A full list if these special characters is given in the *Premier Reference Manual*, under the `search` command, but a few examples will be given here.

`search "^#"`	will match any string of numeric digits.
`search "\""`	will match the quote character.
`search "^& "`	will match any length string of alphabetic characters followed by any space character such as a newline, space, or tab.

In all cases, both of these commands will cause the system variable `found` to be set *true* or *false*, depending on the result of the search, while `foundstring` will take the value of the actual string located.

Searching Backwards.

Premier is capable of searching in *either* direction, and this is selectable using the `direction` command:-

```
direction "backward"
direction "forward"
```

The current search direction may be obtained using the system string variable `direction`, which will have one of the two values shown. If a macro command is created which changes the current search direction, it is good practice to save the current direction, and restore it afterwards:-

```
str "lastdirection" = direction
direction "backward"
search "for whatever"
direction lastdirection
```

This macro may then be used within other commands, without the current direction becomming confused.

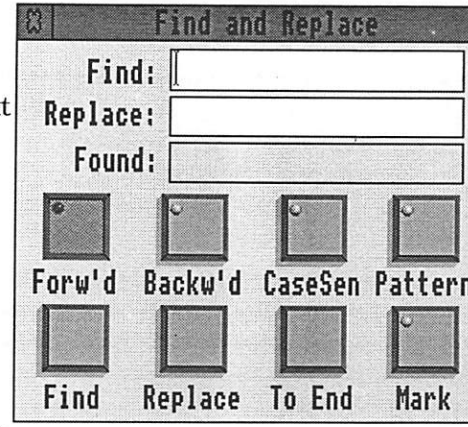
The Finder Dialogue Window.

This is a *dialogue box*, which facilitates the `find` and `replace` actions in an interactive mode, and is displayed by issuing the `finder` command.

This window contains two white text entry boxes in which the string to be found, and any replacement string may be entered, and *action* buttons to initiate the **find** or **replace** operations.

Option selection buttons are provided to select *case sensitive* searching, and to select the *pattern matching* mode. When the latter is selected, the `search`, rather than the `find`

command is used, as described above. The exact string located is displayed in the third text box.



When a search is successful, the **replace** button may be used to replace the found item with the content of the replace box. The **To End** button will repeat the find and replace action to the end of the file.

In *case insensitive* mode, the default, if the string being replaced has a capital letter, then the replacement string will also be given one. By this means, replacements at the beginning of a sentence, for example, will be made correctly. This is not done if *case sensitive* mode is selected.

When a find or search operation is successful, the caret will be moved to the found string, and the window scrolled if required. If the Mark option is selected, the found text will also be marked. It may be necessary to move the dialogue box, if this is obscuring the window.

Searching is carried out in the direction selected by the first two buttons.

Saving and Spooling

This chapter describes how to save and spool files, and the use of the Premier Filer dialogue box.

The Save Command.

This allows any named buffer to be saved to any file name:-

```
save "Letter", "$.Letters.Letter1"
```

This command saves the buffer `letter1` in the file `$.Letters.Letter1`. If the currently selected buffer is to be saved, the system variable `buffer` may be used:-

```
save buffer, "$.MyFile"
```

Spooling.

Spooling is the name given to the action of saving a file as plain text, fully formatted, as it would be printed. In Premier, this also includes dynamic items, just as they appear on the screen. Spooling may be achieved with the spool command:-

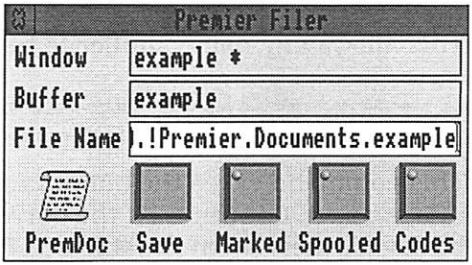
```
spool "$.SpoolFile"
```

A spooled file may be printed at a later date if required, however, it will not include any printer control codes, such as font selection, and will be printed as pure text, including the correct number of space characters instead of tabs, and the appropriate number of newline characters required to page skip, etc.

It is possible to save a file containing printer control codes, if saving is done from the Premier Filer Window.

The Filer Window.

This is a dialogue window, displayed when the command `filer` is given, either from a function key, menu item, or from the command window.



This window shows the currently selected window and buffer names, together with the file name associated with the buffer. If the file was loaded by dragging from a directory viewer, the full path name will be displayed, as shown in the illustration. Otherwise, the file name may be the same as the buffer name.

In either case, the displayed file name may be edited, and the **return** key used to save it to disc.

Alternatively, the displayed icon, showing the file type, may be dragged to the directory viewer where the file is to be stored. In this case the file name will be taken from the *leaf name* of the displayed file name, in this case `example`.

If a marked area exists, this option may be selected by clicking the `Marked` button, in which case just the marked area will be saved.

If the `Spooled` button is clicked, the file will be saved as pure text, as described above.

If the `Spooled` option is selected, it is also possible to click the `Codes` button, to include all printer codes in the output.

Expressions & Operators

This chapter explains how to manipulate strings and numbers, by what are known as *string* or *integer expressions.*

An *expression* is simply a collection of items separated by *operators*, which perform some action on the items, such as adding two integers together, or joining (concatenating) two strings.

Integer Expressions and Operators

These are used to perform calculations, and may include both integer constants, such as '35', and integer variables or functions, both user defined and system types. The usual symbols are used for arithmetic operators, as follows:-

+	Addition
-	Subtration
*	Multiplication
/	Division

These are executed in the conventional order, i.e. multiplication and division *first*, followed by addition and subtraction. Also, parethesis (brackets) may be placed around part of the expression which must be evaluated sepatately. For example:-

```
position = page * ( 3 + line )
```

Here, 3 and line are added first, and the result multiplied by page. There is no limit to the nesting depth of parenthesised expressions.

It is often convenient to write values as *hex* numbers, and this may be done using the form `0x2A`. The '`0x`' pre-fix tells

Premier that the item is a *hexadecimal* number. As with decimal numbers, all characters up to any invalid digit will be taken to be part of the number, to a maximum of 10 characters in decimal, and 8 characters in hex. Upper or lower case letters are permitted:-

```
key 0x4e = insert "N"
```

Integer Conversion Operators.

Other operators may be used to convert integers into strings.

$	Convert the following integer into a string.
%	Convert into a percentage, i.e. divide by 100.
£	Convert to a formatted money string, inc. £ sign.
?	Convert to roman number form.
&	Convert to hexadecimal form.
|	Convert into an ASCII character.

The $ operator simply forms a string representing the integer value. The % operator converts the integer value after first dividing by 100, so that a value in pence may be correctly displayed in pounds and pence.

The £ operator formats the following number into a money string, including the £ sign, the decimal point, and with commas between thousands. The value is assumed to be in pence. The total width of this string is defined by the value of the variable `poundwidth`, which may be set by the user, and defaults to 10 characters, including the £ sign, which is always placed at the extreme left. For example -

```
£123456  => "£ 1,234.56"
```

The | (bar) operator, produces a string containing the single character having the ASCII value of the integer. The following

statements are *not* valid expressions, but indicate how the conversions are done. The => symbol below should be read as *evaluates to.*

```
$37        => "37"
$(a+b)     => "27" ( or whatever! )
%12340     => "123.40"
|0x4D      =>  "M"
```

The ? operator converts the following number to roman numerals, in lower case. This is mainly used for page numbering, etc:-

```
?19        => "xix"
?+1990     => "MXM"
```

String Expressions and Operators.

The complete set of string operators is as follows:-

+	Concatenates two strings together.
#	Convert the string to an integer.
n>	Right justify in width 'n'
n<	Left justify in width 'n'
n=	Centre justify in width 'n'

The first (+) operator simply concatenates two strings together:-

```
"Page " + $page => "Page 32"
```

Here the $ operator described above, converts the current page number into a string, which is then concatenated with the word "Page ".

The hash (#) operator converts the following string into a

number:-

```
#"123"    => 123
#"567F"   => 567
#"XYZ"    => 0
#year     => 1990
```

The last example converts the system string variable year into the equivalent integer value.

The justification operators take the general form:-

```
integer_expression operator string_expresion
```

If the expressions are more than just simple items, then parenthesis must be used, as in the following examples:-

```
10>year         =>  "      1990"
10<$page        =>  "27          "
10="abcd"       =>  "   abcd    "
10>%balance     =>  "    123.95"
```

The last example shows how to format money, in a given column width, where the £ symbol is not required..

```
(pagewidth-10)>"End."  =>    "         End."
```

This last example would of course produce a string ten characters shorter than the defined page width, somewhat longer than shown here.

Non-Printing Characters.

It is often necessary to include characters in a string which it is not possible to type. One way to do this is to use the *bar* (|) operator described above, but where more than one

character is required, special character sequences may be used within a quoted string constant. These are as follows:-

\a	The bell,	ASCII 7
\b	The backspace,	ASCII 8
\t	The tab,	ASCII 9
\n	The newline,	ASCII 10
\f	The formfeed,	ASCII 12
\r	The return,	ASCII 13
\\	The character '\'	backslash
\"	The character ""	quote
\27	The character 27 ASCII	(decimal)
\x7F	The character 7F ASCII	(hex)
\@	The ASCII zero,	NULL.

Examples:-

```
string1 = "This string\nis on two lines."
string2 = "This includes the \" character"

send   "\27E"
insert "\f"
```

Dynamic Text

This chapter describes the facilities offered by Premier to display the equivalent of *spreadsheet slots* within a document. This is done using Premier's unique *dynamic text* feature.

What is dynamic text?

As it's name implies, *dynamic text* is text which may change *dynamically*, that is, to appear differently according to external factors, in contrast to normal text, which, once typed, is fixed until physically edited. Such text has many uses.

There are many occasions when variable items are required in a document. For example, page numbering, the current date, the rate of VAT or other variables within an invoice, or the recipient's name and address in a customised letter or mail shot.

How is it Done.

At it's simplest, a dynamic text item, is a string variable name embedded within the document itself, which is replaced by it's value whenever that part of the document is displayed, saved to disc, or printed, etc. At it's most complex, it can be the name of a command macro, which will be executed whenever required.

It follows that not only can it cause variable items to appear within a document, but can also be made to perform other tasks, using the full power of Premier's command language. Margins and tab settings may be changed at key places, on screen fonts changed, special codes sent to a printer, calculations performed, or information fetched from a data

base. The limit is your own imagination.

A Dynamic item is only recognised and executed with a PremDoc (or postscript) window, and consists of a text string between a pair of up arrow, or *circumflex* characters, such as:-

```
^item^
```

Simple Items.

A simple dynamic item consists of a single string variable, string function, or macro name. In the case of a variable or function, the value of the item will be displayed within the document, at the point it is entered.

In the case of a macro, no text is displayed, but the macro will be executed whenever that part of the screen is refreshed, or the document is spooled or printed out.

While there is no limit to what this macro may do, it should be realised that long complex tasks will delay screen refresh at that point. Simple tasks such as selecting fonts, or resetting tabs or margins, etc, cause no perceptible delay.

Direct Commands.

Where simple formatting commands are required, such as centering or right justifying, etc, there is no need to define and embed macro names to achieve the desired effect. The simplest solution is to enter the command(s) between curly brackets, as a command list;-

```
^{centre}^
```

This example will cause the following text to be centered on the line, assuming there is enough room.

Direct Expressions.

This type of dynamic item is similar to a conventional spreadsheet slot, whereby direct calculations may be entered, and the result displayed within the text. Unlike spreadsheet slots however, they are not restricted by rows or columns, and may be freely placed within the text.

A direct expression must start with an equals sign "=", followed by a string expression, such as:-

```
^=£(vatrate*amount)^
```

This example computes the product `vatrate*amount`, and displays the result as a formatted money string, using the '£' operator. Note that the result must be a string, or it cannot be displayed.

Direct Function Items.

This type of dynamic item consists of a mixture of both types of direct items, consisting of both a command list within curly brackets, and a value signalled by an equal sign. For instance:-

```
^{vat1=vatrate*amount}=£vat1^
```

Since there are no rows or columns, it is not possible for one dynamic item to refer to another item by it's position. However, using this method, which produces the same result on the screen as the last example, allows the value displayed (`vat1`) to be used within other items.

It should be noted that all variables refered to within direct dynamic items, as well as within normal functions and macros, must have been previously defined prior to being used in the definition.

Direct Item Compilation.

Within Premier, all commands, whether entered at the keybaord, read from a command file, or entered as direct dynamic items, are *compiled* into an internal code, for fast execution. This action is automatic, and need not concern you directly, except that you should be aware of what is happening.

When first entered, or first encountered in a newly loaded file, Premier will create either a string function or macro from the dynamic item, and do the necessary compilation. If there is an error at this stage, it will be reported, and the item ignored. If successful, compiled code will be produced, and stored just like any other function or macro, but with a name consisting of the entire item text, between the two circumflex characters.

This is done by entering the required textual macro or function definition into the Journal window, and executing it as though it had been entered in the command window. This action takes considerably longer than subsequent execution of the compiled code, and may cause some delay when a file containing many of these items is first loaded.

This, and other general formatting, is however done in the background, and normal editing may continue. You may however see the hourglass pop up momentarily, as this process continues.

Entering Dynamic Text.

Since the *circumflex* character has special meaning within a `PremDoc` file, and must always come in pairs, it is not possible to enter this from the keyboard, while the caret is in this type of window. There are however two methods by which such items may be directly inserted.

The first is to use the `insert` command, from the command window, function key definition, or menu macro:-

```
insert "^date^"
```

This will directly insert the date, as a dynamic text item, into the document, at the current caret position. This method is often used from a function key, for instance to insert font control items at the touch of a key.

The second method provides an easy to use facility in an interactive editing environment, for entering any dynamic item, by using the `enter` command.

This command, usually also issued from a function key, causes a white text entry box to appear at the current caret position, with the caret positioned inside it, ready for you to enter the dynamic item. This should **not** include the *circumflex* characters, which are added automatically for you in this instance. When the **Return** key is typed, the item will be inserted, and the box removed.

Editing Dynamic Text.

Whenever the caret is positioned at the start of a dynamic item, and then moved to the *right* over it, the same white text entry box will appear at the caret position, displaying the dynamic item name. The caret is not immediately positioned

inside this box, so that moving it further right will cause the box to be removed, and the caret to be placed at the end of the displayed dynamic text, if any, so that continuous cursor movement is not interupted.

Where several dynamic items have been inserted adjacent to each other, the caret may show each in turn. In this way, the caret may be positioned where required, so that text may be inserted between the adjacent dynamic items. Note that where the dynamic item does not itself cause text to be displayed, such as would be the case with formatting commands, etc, the caret will appear to remain unmoved, whichever side of the dynamic item it actually resides.

To edit the dynamic name, while the box is displayed, the `enter` command is again used, to cause the caret to enter the box. Alternatively, the mouse **select** button may be ckicked over the box. Within the box, the required editing may take place as within a normal dialogue box entry window, or the name deleted completely, before typing **Return**. In the latter case, when **Return** is typed the box and the dynamic item will be removed from the document.

Font Control.

Dynamic items which control fonts are usually command *macros*, which do not of themselves cause any text to be displayed. Consider the following example:-

```
macro "FB"
{
     font = bold
     if ( fontbold )
          send PFB1
     else
          send PFB0
}
```

This macro first issues the `font` command which *toggles* the bold font condition. It then uses the system variable `fontbold` which has the value *true* if bold font is now turned on. If this is the case, the `send` command is used to send the string variable PFB1 to the printer. If `fontbold` is *false*, the string PFB0 is sent. These printer code strings, containing the codes to turn bold font on and off respectively, are normally defined within a printer command file, and executed at program start up. (See the chapter on printing.)

Coloured font may also be selected, and if your printer does not have this facility, no `send` command need be included:-

```
macro "CR" { font red }
```

Alternatively, this may be entered as a direct command, as described above.

Using Functions.

Where the same calculation may be required in many places, it is more efficient to define a function to perform the calculation, rather than repeat it within direct expressions. Consider the following;-

```
int function "VatDue" = IP(1)*vatrate/100
```

This function will compute the VAT due in any amount passed to the function as a parameter. It may be used within a direct dynamic calculation, as in the following example:-

```
^=£VatDue(amount)^
```

This will display the VAT due on the value of `amount`, assuming it has been defined, formatted as a sum of money.

Headers and Footers.

Another use for dynamic text, is to define *headers* and *footers* for a document.

These are dynamic text items inserted automatically by Premier, in any of the available page header and footer margins. To do this Premier will look for items having the names HEADER or FOOTER, or HEADER1, FOOTER3, etc. The numeric suffix indicates the margin line on which the item is to appear, numbering downwards from the top for headers, and upwards from the bottom for footers. Items without numbers are displayed on margin line zero, that is on the top and bottom lines of the respective page margins.

You may override these pre-defined names, to avoid confusion between documents, by using the commands:-

```
header = "HeadName"
footer = "FootName"
```

These names will then be used in place of HEADER and FOOTER described above.

The following example defines a page footer displaying the page number, justified to the right margin.

```
str function "FootName"
         = rightmargin>("Page "+$page)
```

This uses the string justification operator '>' to format the page number string at the right hand end of a string of width `rightmargin`.

Note the use of parenthesis (brackets!) which ensure that the whole string is right justified, not just "Page ".

Document Formatting.

As stated above, a *macro* may contain any Premier commands, and these can include tab and margin setings. For example:-

```
macro "IndentOn"
{
     tabs 10,20,30
     margins  leftmargin + 5, rightmargin
}
```

This macro first re-defines the first 3 tab positions, and then re-defines the left margin to be indented by five character positions. The system variables `leftmargin` and `rightmargin` have the values defined for the whole document using the `page` command, and may be used in another formatting macro to terminate the indented section.

```
macro "IndentOff"
{
     margins  leftmargin, rightmargin
}
```

Such macros would be defined in a command file, having the same name as the document, and stored in the `Commands` sub-directory, to be executed automatically whenever the document is loaded.

Nesting Dynamic Items.

Dynamic items may be nested, that is, inserted into other dynamic items. One possible use for this is to include font controls into header and footer items, but other uses will be apparent. Consider the following:-

```
str function "fred"
      = "^CR^The time is now ^time^^CBk^"
```

This defines a string function, called `fred`, which may be used to display the string shown. This string itself includes dynamic item macros to define red font, and the system variable ^time^. In cases like this, it is important to include the circumflex characters, which identify the word time as a dynamic item. Do not forget to re-select black font, before the string end.

Data Base Facilities.

This chapter describes Premier's many powerful commands for reading, writing, editing, searching, and gerally maintaining data base files. It includes a description of the data base *dialogue window*, which facilitates most functions, in an interactive environment.

What is a Data Base.

From Premier's point of view, a data base file is simply an *organised* text file, containing any information which can be expressed in textual form. The data is organised into *records*, each containing one complete set of data, somewhat equivalent to a single card, in a card index file. Records are separated from each other by the *newline* character, (ASCII 10).

Each record may be split into a number of *fields*, each containing a specific item of data, such as a name, the date, or an invoice number, etc. Fields are separated within each record by a *comma*.

The only special requirement, is that the first record of the file should contain the names, or titles, of each field, and not real data. Such titles should be single words without embedded spaces, and should be padded to the required length with spaces, up to the separating *comma* or final *newline* character.

Such data files are manipulated directly on the disc, and not loaded into memory, so that data files larger than the available memory may be used.

Compatibility.

The format described above is *Comma Separated Value*, or *CSV* format, and is compatible with the output obtainable from a number of data base programs. If you have a program with the capability of outputting data in this format, transfering such data into Premier is a simple matter.

If you require Premier to successfully modify such data, it is important the all equivalent fields are the same length, and that all records are the same length. Premier is capable of reading data files having different length fields and records, but clearly it cannot overwrite a short field with a longer one.

The only conversion likely to be required, may be to add the field names in the first record, if these are not provided, and this may be done using any text editor, including Premier. Simply create a `PremData` window and drag the data file into it. This type of window has a maximum width of 1000 characters, the maximum record length allowed in Premier.

Opening a Data Base File.

A data base file may be opened for reading and writing using the command:-

```
data "FileName"
```

Or, as explained in an earlier chapter, by dragging it's icon onto the **Premier** icon on the icon bar.

When either of these things is done, Premier will first close any data file already open, then check to see if it has previously opened the new file. If it has, it will re-open it and re-read the last record accessed within it. If it is a completely new file, it will open it and read the first record.

As explained above, the first record should consist of the *field names*, and these will be read, and the field lengths noted. Premier will then automatically set up *string variables* having the same names as the fields, enabling you to refer to these in the same manner as any other variable. The first real data record will then be read, and the field contents assigned to the corresponding variables just created.

Reading Successive Records.

To read through the data base, one record at a time, it is only necessary to issue the read command. **Premier** will then repeat the above action on the second, and successive records, assigning the field contents found to the appropriate variables, replacing the previous values.

At any time, you may re-assign one of the field variables to a new value, and write this back to the file, using the write command:-

```
write
```

The string values assigned to these variable by **Premier** will always have any trailing spaces stripped, and will be padded to the correct lengths when written back. If you have assigned a string which is too long, this will be truncated, and a warning message given. The field lengths are available, as described later.

Searching for Records.

Rather than stepping through the file examining each record in turn, we can ask Premier to do this for us, using the seek command, which has the form:-

```
seek  fieldnumber "string"
```

When this command is given, Premier will search, from the next record, for one having the string "*string*" in field *fieldnumber*. This search is **not** case sensitive, and will find any field *containing* the given string, for example:-

```
seek 2 "jones"
```

This command will find "Mr. A. JONES", etc, in field 3.

Using Filters.

A *filter*, in this context, is a device which only lets through data records which meet some pre-defined set of conditions. When a filter has been set, Premier will skip over records which do not meet the conditions, or do not *come through* the filter, as though they were not there. In this way, you can send mail shots to selected groups of people, or obtain a print out of customers with overdue accounts.

There is no limit to the number of filter conditions which you can apply to a data file, and you can apply zero or more conditions to any field in the records. A filter may be set with the filter command, having the following form:-

```
filter <field> <logic> <operator> <string>
```

These items represent respectively, the *field* the filter is to be applied to, a *logical action*, and the *comparison operator* to be applied between the actual field content, and the *data string*. An example may clarify this:-

```
filter  4  and  equal  "wilts"
```

This sets a filter on field 4, assumed to contain county names, and lets the record through if this field is equal to the string "`wilts`". The logical action *and*, affects the way this

filter if treated relative to any others.

With the *and* action set, this, and all other filters having the *and* action, must all be *true*, before the record will pass through the filter.

If this logic action had been *or*, the record would be passed regardless of any other filters which may be set. For example:-

```
filter 4 or equal "Hants"
filter 4 or equal "Wilts"
filter 3 and lessthan    "18"
filter 3 and greaterthan "15"
```

If these four filters are set, and assuming field 3 contains pupils ages, all records of pupils living in Wiltshire or Hampshire, over 15 and under 18 will be selected. In other words, at least one of the filters with the *or* logic action must be *true*, assuming there are any, in addition to all the filters having the *and* action.

All filters on a specific field may be cleared, by including only the field number, thus:-

```
filter 3
```

This, in the above example, will remove both the age conditions from the filter set on field 3

All fields number from 1, and out of range field numbers will provoke an error message.

Indexing.

So far, all file reading has been done by examining all records on the disc, in whatever order they appear. It is often necessary to obtain say, a list of names in alphabetical order, or to be able to access a series of records as fast as possible. To do this we may create an *index*, using the command:-

```
index  fieldnumber
```

This will cause the data file to be scanned from the beginning, and a sorted list formed using the field number given. This list is held in *memory*, and may be searched very fast. The index is preserved when the data file is closed, or a new one opened, and may be re-used by re-openning the same data file.

Each `read` command will now read the next record from the disc in indexed order.

If a filter has been set before using this command, then the index will *only* contain records which pass through the filter. The `seek` command, if used specifying the *same field* as the index, will now search the sorted index in memory, to locate the required record fast. The entire record identified will then be read from disc in the normal way.

It should be noted that a search made on any other field will require Premier to read each record from disc, and since this will still be done in sorted order, may actually take *longer* due to the random disc accesses.

Records read via the index are still required to pass through the filter, which they will do, unless the filter has been changed since the index was formed. In this case a further filtering will take place on the indexed records, which may

provide further selection facilities.

Finally the `rewind` command will set the current data file to the beginning, either to the first record, or to the first index entry if an index exists.

Special Field Variables.

In addition to the string variables set up according to the field names, Premier provides a special *system variable*, which may be both read and written to, called `fieldvalue`. In order to specify which field is required, this should be added in parenthesis, as in the following examples:-

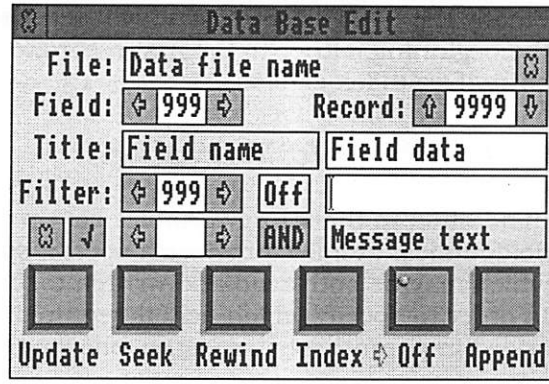
```
fieldvalue(4) = "This is new data"
str "savedfield" = fieldvalue(6)
```

The first command sets field 4 to have the value shown, and is equivalent to using the field name variable. The second command creates a new string variable which is then used to store the value of field 6. This system allows fields to be directly addressed by number, which is useful in many instances.

We may also obtain the name of any field using the system variable `fieldname(n)`, and the length of any field using `fieldlength(n)`.

These are used in a similar manner to the above, except that they cannot be assigned a new value. Lastly, the system variable `fieldcount` provides the number of fields in the current data base records. An example macro using these variables in given in the command file chapter.

The Data Base Dialogue Box.



This window may be displayed by giving the command `datawindow`, and provides easy access to all the data base facilities described above. This is a multi-function control box, with many possible combinations of actions.

At the top of the box is a text window showing the currently selected data file name, with a *file close* icon to the right.

Below this are the record and field selectors, and the use of these should be fairly obvious. Clicking on the *up* or *down* arrows, moves the current record backwards or forwards through the file, while the *left* and *right* arrows select fields in the corresponding directions. In each cases the record or field number is indicated in the white box between the arrow icons. Records will be read via the index, if one has been created.

On files having *variable length records*, it is not possible to step backwards through the file unless an index exists, and in these cases the up arrow icon will be blanked out.

The name of the field selected, and the actual field content, are displayed in the two areas directly below the record and field selectors.

The next two rows of icons allow the setting of filters. Any number of filters may be set, and may apply to any or all fields within a record. A filter consists of a *field number*, some *textual data*, an *operator*, and a *logical action*. The operator defined will be applied to the data and the specified field content, to decide whether or not to allow a record through the filter.

These operators are shown in the indicator, using the charater symbols listed in the following table..

lessthan	<
greaterthan	>
equal	=
notequal	#

The *logical action*, either AND or OR, is shown in an adjacent box, and may be toggled by clicking on it.

The *filter selector* operates in similar fashion to the field selector described above, except that the left arrow icon always returns the indicator to zero, and in this condition, no filter data is displayed. The indicator should always read zero before attempting to define a new filter.

Setting Filters

First click in the white data window to the right of the selector, and type in the data to be used, for instance a persons age, or town, etc. Next select the operation on the lower selector, by clicking the right arrow icon until the required operator is shown, i.e. either '>', '<', '=', or '#', as described above. Then the *logic action* may be toggled between AND and OR, by clicking on it. Finally, ensure that the *field* required is selected as described above.

All we have done so far is to set up the data required for the filter, and this may be modified if it is not yet correct. To actually set the filter, the *tick* icon should be clicked, and when this is done, the word 'Set' will be shown in the filter status box in place of 'Off', and the filter number will show '1'.

To set another filter, on the same or a different field, click the left arrow to select zero on the filter number display, set the data as above, and again click on the 'tick' icon. Note that the new filter will now be number one, while the previous filter will become number two. i.e. all new filters are added at the front of the list.

You can modify any filter, by finding it using the right arrow icon, modifying the data displayed, and again clicking on the *tick* icon. Since the filter number display does not now show zero, the indicated filter will be updated, rather than a new filter being created.

The *cross* icon next to the tick, may be used to cancel the filter indicated, or the *first* filter if the display shows zero. All filters may be removed by repeated clicking on this icon, with zero displayed, until the status box indicates 'Off'.

Having set the required filters, stepping through the file using the record selector, will only display those records which pass the tests set in the filters. Similarly, creating an index will produce an index consisting only of records which pass the filter tests.

The Action Buttons.

The *Index* button issues the `index` command, which is fully described above. It is possible to toggle the index on and off, by pressing the adjacent button.

The *Rewind* button executes the `rewind` command, putting the current record to the start of the file, or to the first index entry.

To *Seek* out a particular record, enter the required search string into the *filter data box*, and without typing Return, press the *Seek* button. The seek operation will be performed on the currently selected field. Only records which have been indexed and pass through any filter set will be examined.

Entering & Editing Records.

This may be easily done using the dialogue box. First set the field indicator to 1, and click on the *field data box* to position the caret. If this box contains data, it may be edited, or removed and re-typed as described earlier. When the entry is correct, type Return. This will assign the new data to the appropriate string variable, and select the next field.

It is also possible to move between fields using the up and down cursor keys.

When the all fields have been entered or corrected, the new data may be written to disc, either by pressing the *Update* button to correct the current record, or by pressing the *Append* button to write the new record to the end of the file. If an index exists, this new record will be added to it, if it passes through any filter set.

Printing

This chapter describes how to print documents, and how to send control codes to your printer. A knowledge of the preceeding chapters is assumed.

It should be clearly understood that Premier does not print documents in *graphics dump* mode, by sending *bit-images* to the printer, as is done in desk top publishing programs, or in !Paint and !Draw, but uses normal *text mode* printing. Thus where special fonts are shown on the screen, control codes must be sent to select the apropriate font on the printer. This method provides flexible access to all the facilities offred by your printer, such as the selection of high quality or draft mode printing, foreign language fonts, etc.

This applies whether you print a document using the `print` command, or by dragging icons onto a printer driver on the icon bar. In the latter case, the printer driver requests Premier to do the printing, and Premier responds by creating a suitably formatted *text file* containing the necessary control codes, informs the printer driver of the file name, and leaves it to get on with the job.

The `print` command sends the currently selected buffer to the printer, exactly as it appears on the screen. It may be followed by two optional parameters, specifying the number of copies, and the page numbers required. If these items are not provides, one copy of the whole document will be printed. e.g.:-

```
print  2,  5, 5
```

This will print two copies of page 5 of the current document. For full details, consult the *Premier Reference Manual.*

Printing may be aborted at any time by using the **escape** key.

Single Sheet Printers.

If your printer takes single sheets of paper, and has no sheet feeder, Premier may be made to pause between sending each page to the printer, by using the pause command before printing the document. This takes the form:-

```
    pause on
or  pause off
```

Hidden Text

Premier has the unusual feature of being able to show comments or other text on the screen, which will not be printed out. This is done using the `hide` command, within dynamic items either side of the text to be hidden. The text so marked will appear grey on the screen, as a reminder.

This command takes the form:-

```
    hide on
or  hide off
```

Suitable direct dynamic items would therefore be:-

```
^{hide on}^   and  ^{hide off}^
```

Control Codes.

These are sent to the printer using the send command, which takes the form:-

```
send "string"
```

The string may be a literal as above, or more usually a string variable. The command cannot be given from the command window, but must be included in a dynamic text item, embedded in the document at the point in the text where the codes are to be sent.

Whenever this command is encountered within a dynamic item, it has no effect if printing is not currently taking place.

Initialising the Printer.

Two special string or function names are recognised by Premier within a printing context, and used in special circumstances. The first of these is the name `PIN`, standing for Printer INitialise.

If this item has been defined, the string represented is always sent to the printer at the start of every print job, and may be used to set the printer into what ever mode is required. If the string is not defined, nothing is sent, and no error results.

Printing Pound Signs.

The second special name is POUND, and may be used to overcome the ubiquitous problem that the pound sign is not part of the default US ASCII character set, and different printers require different methods to print it.

The pound sign is only available if the U.K. character set is used, in which case it replaces the # character, (ASCII 35). In addition, within the Archimedes computer, this sign is given the value 163, in accordance with the ISO Latin 1 standard font, which many printers do not understand.

To solve this problem, Premier treats the £ sign in a special way during printing. When this character is found, Premier

looks to see if there is a *string variable, string function,* or *macro* called POUND (upper case) defined, and if there is it sends the string represented to the printer, or executes the macro, instead of sending the £ character to the printer. If no such variable is defined, the £ character (163) will be sent as usual.

If pound signs do not print correctly on your printer, consult the printer manual and find what code sequence is required. This may mean, for instance, making a temporary switch to the U.K. character set, sending the # character, and switching back. Such a macro may look like this:-

```
macro "POUND" {send"\27R\3#\27R\@" }
```

This sends the sequence 27,82,3,35,27,82,0. Note the use of `'\@'` used to send the null character. See the chapter on *String Constants*, for full details of how non-printing codes may be embedded in text strings.

User Defined Menus

This chapter describes how you can define your own screen menus, to customise Premier for your own purposes. If your Premier *application* provides on screen menus, these will be defined within the Premier command file defining the application, and is totally under your control.

The menu available from the Premier icon on the icon bar is defined within Premier itself, and it is not possible to modify it.

Menu Definitions.

A menu may be easily created using the `menu` command, which has the following form:-

```
menu "mymenu" = "Action1,Action2,Action3"
```

This creates a menu called `mymenu`, and this name will appear as the menu title when it is displayed. The options will be displayed as listed in the string, and this may be a *string constant* as above, or any *string expression.*

Each of the options may be the name of a *command macro*, which will be executed when that option is chosen from the menu, or the name of another defined *menu*, which will be offered as a sub-menu when selected.

Additionally, the following special characters may be used to pre-fix the macro names, and will have the following effects:-

>	Provide access to a dialogue box.
!	Display a tick by the option.
-	Fade the option, (non-selectable).

In addition, the *bar* '|' character may be used *in place of* a comma between the options given, and in this case the options will be displayed separated by a dotted line. For example, the following menu definition, will produce the menu shown in the illustration.

```
menu "Save" = ~Marked,!Spooled,ToFile|Quit"
```

When *faded*, as in all RISC OS menus, the option will *not* be selectable. In the example, this would apply where there was no marked section to save. A *tick* is used by convention to indicate a currently selected option, and the *dotted line* simply to separate unrelated groups of options.

The '>' is used where the option chosen causes a dialogue window to be displayed, such as the filer window, and allows this to be done by simply moving the pointer over the '>' character, rather than clicking on the option.

No check is made to see if a macro exists when the menu is created, but an error will be created when the item is selected.

The faded and ticked attributes may be applied to menus already created by using the flag command:-

```
flag "Save",.3, true, false
```

This would have the effect of ticking the third item, in the above menu. See the *Premier Reference Manual* for full details of this command.

Displaying Menus.

This is done using the command:-

```
offer "Save"
```

This command, usually assigned to the mouse menu button, will cause the menu to be displayed on the screen. This may be included in a macro which creates and offers a specific menu, according to which window was clicked over, and including whatever flag options were appropriate.

Sub-Menus.

A sub-menu is created in exactly the same way as any other menu, and may be offered in the same way. To offer it as a sub-menu, it's name should be included in *place of a macro* name in the parent menu. For instance, the Save menu in the above example, could be offered as a sub-menu as in the following example.

```
menu "Main" = "Open,Save,Close,Print"
```

When this menu is offered, Premier will detect that `Save` is another menu, and offer this when that option is chosen. The `'>'` character will be added automatically, to provide the normal sub-menu selection mechanism.

You should avoid creating infinite loops of sub-menus!

Other Uses for Menus.

Menus may also be used to allow the user to select from a list of text items, using the `choose` command. This is similar to the `prompt` command, both of which are fully described in the *Premier Reference Manual*. The general form is:-

```
choose "List", variable
```

This will cause the menu "List" to be offered, and the item chosen to be assigned to the variable given.

If the variable is a string type, the text displayed in the menu item will be assigned to it. If an integer variable is given, the text displayed will be converted to an integer if possible, before being used. If no leading digits are found, a value of zero will be used.

A useful system function in this context is `content`, which will provide a list of files within a named directory, separated by commas, as required by the menu command. For instance:-

```
menu "Files" = content("$.Letters")
```

This will create a menu offering all the files within the named directory, and may be used with the choose command:-

```
choose "Files", filename
```

This assumes that `filename` is a defined string variable.

Formatting the Date

This Chapter describes the `date` and `time` system variables, and how you may define your own formats.

Date Variables.

There are two *system variables* which provide the date, and each is intended for a different purpose. These are data and stddate.

The date variable, already introduced, is intended for use as a dynamic item within documents, where the current date is required. The format of this string is variable, as described later.

The date variable `stddate`, is of fixed format, and is arranged so that the most significant digits appear first. This is so that the string may always be used with the comparison operators, such as <, >, etc, and produce the correct result. This is mainly intended for use within data base files, where date searching may be required.

The Time and Year.

The system variable `time`, produces the time in 24 hour format, while `year` supplies the current year in 4 figure format. Both of these are obtained from Archimedes system variables, and are also of fixed format.

The Date Format String.

The display of the date, using the system variable `date`, is defined by the value of the system string variable `dateformat`, which is user definable. It consists of *normal* characters which are reproduced in the date string exactly as

given, and special *formatting characters*, which define which date elements you require, at the position you want them. These special characters must begin with the % character, which is then generally followed by two formatting characters.

The Special Format Characters.

%24	The current hour in 24 hour clock format.
%12	The current hour in 12 hour clock format.
%MI	The number of minutes past the hour.
%SE	The number of seconds past the minute.
%CE	The number of centi-seconds (hundredths).
%AM	The letters 'AM' or 'PM' as appropriate.
%PM	The letters 'AM' or 'PM' as appropriate.
%WE	The day of the week string, in full, e.g. Tuesday
%W3	The first three characters of the weekday.
%WN	The weekday's number, Sunday = 01.
%DY	The day of the month, e.g. 01.
%ST	The suffix 'st', 'nd', 'rd', or 'th' as appropriate.
%MO	The month name string in full, e.g. August.
%M3	The first three characters of the month.
%MN	The month number, January = 01.
%CE	The current century, i.e. 19.
%YR	The current year, i.e. 89.
%WK	The week (Mon-Sun) number in the year.
%DN	The day number in the year, i.e. 048.
%%	The explicit character '%', i.e. not a flag.

For example, if the string should include %MO then, in the date string, these characters would be replaced by the current month name in full. A complete list is given on the next page.

All numbers are in fixed field width format, including leading zeros where necessary. These may be suppressed by pre-fixing the letter code with a 'z'. i.e. the format string "zdy" will produce 7 rather than 07. All codes may be in upper or lower case. For example, suppose we wanted the date to appear as-

```
"Today is the 16th of June, in 1990"
```

This could be achieved using for format string:-

```
"Today is the %DY%ST of %MO, in %CE%YR"
```

Here, the sequence %DY will be replaced with '16', %ST with 'th', and %MO with 'June'. The year is formed by %CE as '19' and %YR as '90'. Any other characters are included as given.

To set the format string to the value required, as above, simply make a normal string assignment:-

```
dataformat = "%DY%ST %MO %CE%YR"
```

This would produce the date "16th June 1990"

Command Files.

This chapter has been left until last, as a knowledge of the preceeding chapters is required to make full use of Premier command files. It describes how to create command files, what happens when they are executed, and gives some examples of macro which demonstrate some of the techniques available.

However, the possible uses for command files is so vast, that we can do no more than whet your appetite, and point you in the right direction. Your immagination is the limit.

The Premier command language is entirely *free format*, and commands may spread over several lines. The example shown here are formatted having various sections indented, as ~~ant~~ this style makes for easier reading and understanding. Comments may be included using the '!' character. All text following this, up to the end of the current line, will be ignored.

Command File Execution.

A Premier command file may be executed by using the `execute` command stating the file's name, by double clicking on it's icon within a directory viewer, or by dragging it's icon on the Premier icon on the icon bar. Whichever method is used, the result is the same. A forth method using of the `do` command, is slightly different, and is discussed later.

First, Premier will look in the directory `<Premier$Dir>.Object` for a file of the same name, and if it finds one it will compare it's date with that of the command file.

If the *object file* is of a later date, this will be loaded and executed. This file is assumed to contain a compiled version of the command file, in Premier's own object code, and may be executed very much faster than is possible with the textual command file. This is firstly because all syntax checking has already been done, and the file is known to be correct, and secondly because the various commands and operators etc, have been looked up and converted to simple codes, ready for immediate execution.

If the command file has a *later* date than the object file, it is assumed to have been modified, in which case the object cannot be used as it no longer represents what is in the command file. Premier will therefore load and execute the command file directly.

This process may discover syntax errors, where unknown commands are found, or the wrong data is supplied to known commands, etc. These errors will be reported as they are found, and the operation will be abandoned.

As this process continues, object code will be produced, and executed. Assuming all is well, Premier will finally write this object code back to disc, for use next time.

Since the command file is actually executed *incrementally*, that is, *statement by statement*, it will be creating variables and macros, defining keys, etc, as it goes. If an error is subsequently found, these actions will not be undone.

As described in another chapter, if the *journal window* is not open, errors will be reported in the Premier message window. However, they will also be written to the journal buffer, and may be viewed at any time. These journalled messages will include a copy of the offending line, and an arrow indicating the error position. This position will be the farthest Premier could get, after trying all possible commands and variants.

The actual error may be earlier than indicated if Premier erroneously interpreted the error as part of an alternaive command.

Creating Command Files.

These are easily created using the icon bar menu as described in an earlier chapter, and should be saved into the `Commands` sub-directory.

Document Format Files.

Whenever a document file (of type `PremDoc`) is loaded, by whatever means, Premier will look in the `Commands` directory for a command file having the same name. If it finds one it will be executed, as described above. No error is produced if no such file exists.

This mechanism is provided so that you may define any formatting macros, variables, or other items required by the document. It may also contain commands to open data files, etc.

Since these files may include the `execute` command, it is quite possible to create say, a general letter format file, and have this executed from a number of different command files, all of which are to share the same format.

Premier also maintains it's own formatting command files, again having the same name as the document, but contained in the `Commands.Formats` sub-directory. These files should not be modified by the user.

Application Command Files.

These are no different to any other command file, except that they would normally contain things like macro definitions, and possibly assign some of these to function keys, etc.

Since double clicking on a Premier command file will cause Premier to be loaded, if it is not already, and the command file to be executed, a specific Premier *application* may be started in a similar manner to any RISC OS application. In addition, since the command file may include the `quit` command, it may carry out some pre-arranged task, and quit without opening WIMP windows, etc.

Tried and tested application command files should be stored in the `Commands.Applics` sub-directory, where Premier will see them, and offer them when the `Applics` option os chosen from the icon bar menu.

User Defined Applications.

Certain conventions have been defined, so that all Premier applications conform to the same standard and behave in the same way, and all applications designed for general release or publication, should conform to this standard, which ensures the following behaviour.

When any one application is loaded, it should create a single top level menu, invoked from the mouse menu button, and having the same name as the application.

When a second or subsequent application is loaded, a new top level menu called `Premier` should be formed, offering as sub-menus all the main menus from all the loaded applications.

This action may be achieved by including the following code at the end of the command file, replacing ApplicName as appropriate.

```
if ( defined( "MainMenuList" ) )
{
     MainMenuList+= ",>ApplicName"
     key 257 = offer "Premier"
}
else
{
     str "MainMenuList" = ">ApplicName"
     key 257 = ApplicName
}

menu "Premier" = MainMenuList
```

The '>' symbol before ApplicName is only required if this is a macro mame which invokes the required menu, and not if it is itself a menu name. If MainMenuList is defined, the new application name is added to it, and the menu button defined to offer the Premier menu. Otherwise a new variable is created, and the button defined to offer the application menu.

Some Useful Macros.

The following represent just a few techniques which may prove useful in your own command files. They are given in good faith, but no guarantee is given that they will meet any exact need. Most assume that the variables used, etc, have been created prior to the macro definition, and may require expanding to include error trapping, or other enhancements.

The first provides a means of automating data entry for a data base file.

A Data Entry Macro

```
macro "RecordEntry"
{
     int field = 1
     while ( field < fieldcount )
     {
          prompt "Enter " + fieldname(field),
              fieldvalue(field)
          field += 1
     }
     append
     update
}
```

This macro makes use of the system field variables to prompt the user to enter each field in turn, until all the fields have been entered. i.e. when the integer field becomes greater than the number of fields in the record, as given by the system variable `fieldcount`.

When this has been done, the new record is appended to the data file, and the screen updated. It is assumed here that a data file has been opened, and that some document window is open, displaying data from the file.

A similar macro could be written to allow prompted searching for particular records, and offer the user the option to amend the record, after displaying it in a suitable window.

The second example demonstrates one possible way of constructing a relational data base function, and is intended only to explain the general method. It is based on the principal that one field in more than one data base file, will contain the same information, and act as a link field between the several files. This example assumes two files, a name and

address file, and an accounts file, are linked by fields containing the customer's name.

A Relational DataBase Example.

```
str function "BalanceDue"
{
     if( AccountName == CustomerName )
          balance = FinalBalance
     else
     {
          dfile = datafile
          data accountsfile
          seek CustomerName, 3
          if ( found )
               balance = FinalBalance
          else
               balance = "Not found"
          data dfile
     }
} =  10 > %(#balance)
```

The string function `BalanceDue` is included as a *dynamic item* in an invoice, or similar document. The value it displays must be the field `FinalBalance`, taken from the data base file `accountsfile`, while the customer's name `CustomerName` is held in another data base file, assumed to be currently open when the function is called.

The `CustomerName` is compared with that last name read from the accounts file, `AccountName`, and if they are the same, then the last record read from the account file must already be the correct one, and balance is assigned the appropriate value directly. Otherwise the correct record must be found from the accounts file.

This is done by first saving the current data file name in the variable `dfile` before opening the account file. The `seek` command is used to find the correct record, having the customer's name in field 3, and if this is successful, the variable `balance` is assigned the required field value `FinalBalance`. Lastly the original data file is re-selected, and the string function itself defined as a right justified string.

This example illustrates an important point, that fields containing the same data, in different files, should have different field names, or confusion would result. i.e. `CustomerName`, and `AccountName`, in this example.

The last example demonstrates a text editing command file, which may be used to replace all `Return` (ASCII 13) characters in a file with `newline` characters (ASCII 10), as required by Premier.

Simple Text Processing.

```
macro "Convert"
{
     start
     repeat
     {
          find |13
          if ( found ) replace newline
     } until not found
}
```

This begins by moving the cursor to the start of the file, then enters a `repeat` loop. This searches for a *Return* character (|13), and if it finds one, replaces it with the *newline* character (a system variable). When no more *Return* characters are found, the loop terminates.

Debugging Command Files & Macros.

It is usually true that even simple macros will have some errors in them, especially while you are learning . The simplest way to test a command file, is to enter each macro into a new command window, and execute it where it is, by giving the do command.

This will execute the content of the window, and report any errors found. If it is successful however, no object file is produced. When all the required functions and macros have been created without error, they may be combined into one command file, and saved to disc. This file may then be compiled into an object file, by dragging it onto the Premier icon or double clicking on it in the normal way.

There are no short cuts to de-bugging command files, and in the end, experience counts. A useful tip, is to select *verbose* message reporting, and to insert the `show` command at places where problems are suspected, to produce useful feedback on what is happening.

A common cause of trouble, is spelling mistakes in variable names, or attempting to assign a string to an integer variable, etc. Watch out also for missing '|' brackets terminating command lists.

The Config File.

A command file by this name is conventionally used to define any default settings required, or to specify applications to be automatically loaded, and may be executed by including it's name on the Run command line in the `!Run` file in the `!Premier` directory.

Index

!	60
-	91
'!'	99
£	60
£ sign	60, 89! 91
!System	3
#	61
$	60
%	60
&	60
+	61
.	10
<	61
=	61
>	61, 91
?	60

A

A Data Entry Macro	104
A Relational DataBase Example.	105
About this Manual	2
Addition	59
An Example Macro	43
and	36, 79, 83
anti-aliased fonts	10
Append	85, 104
application	22, 30, 102
Application Command Files.	102
Applics.	18
arithmetic operators	59
arrow icon	15
assign	29

B

background colour	19
backspace	63
backward	55
bell	63
bit-images	87
buffer	27-28, 31
buffer name.	27, 53
buffers.	27
bug	45
buttons	14

C

card index file	75
caret	53, 70
case insensitive	56
CBk	74
centre	66
choose	94
Circle	2
circumflex	66, 69
ClipBoard	11, 18, 20, 50-52
Closing and Opening Windows.	10
code sequences	9
Codes	58
colour sub-menu	19
Colours.	19
comma	75
Comma Separated Value.	76
Command	8
Command File Execution.	99
Command Files	1, 9, 24, 30, 50, 99
Command Formatting.	39
command language.	21
Command Lists.	37
command macro	91
command window	21
Command.	19
Comments	99
comparison operator	78
Compatibility.	76
concatenating	59
Conditional Commands.	35
control codes	57, 87, 88
Controlling Message Reporting.	46
copy	52
Copying Marked Areas.	52
CR	74
Create.	17
Creating Command Files.	101

CSV format, 76
current buffer 53
current date 65
current direction 55
current record 82
cursor 28, 30-31
cursor keys 13
cursor position 28
cut 51
Cutting Marked Areas. 51

D

data 105
data base 76
data base facilities 82
data base file 75
data record 77
data records 78
dataformat 97
datawindow 82
date 22, 69, 95
Date Variables. 95
dateformat 23, 95
de-bugging 107
Debugging Command Files & Macros. 107
deletedchar 37
diagnostic 47
dialogue box 56
dialogue boxe 13
dialogue boxes 14-15
dialogue window, 75
Direct Commands. 66
direct dynamic item 68
Direct Expressions. 67
Direct Function Items. 67
Direct Item Compilation. 68
direction 55
Displaying Menus. 93
dispose 47
Division 59
do 99
Document Format Files. 101
Document Formatting. 73
document type 18
document types 17
dynamic item 69, 74
dynamic name, 70
dynamic text 65

E

edit 30-31
Editing Dynamic Text. 69
Editing the !Run File 5
either 83
else 105
enter 69-70
Entering & Editing Records. 85
Entering Dynamic Text. 69
Entering Simple Commands. 22
equal 79, 83
Equal to 36
ERROR 20, 45-46
error level 46
ErrorLog 48
errors 25, 100
execute 99
execute command 101

F

faded 92
false 35
FATAL 20, 45
field 78, 80, 106
field data 85
field lengths 77
field names 77, 106
field number, 83
field selector 83
field selectors. 82
field variables 104
fieldcount 81
fieldlength 81
fieldname 81, 104
fieldnumber 78
fields 75
fieldvalue 81, 104
File Saving. 10
filer 5, 10

Filer dialogue box 57
Files, Buffers and Windows. 27
filter 78-79
filter selector 83
filter tests. 84
find 53-55, 106
Finder 54-55
first record. 76
flag 92
font 70
Font Control. 70
fontbold 71
footer 72
footers 72
foreign language fonts. 87
formfeed 63
forward 55
found 36, 38, 105
free format. 99
function key 54, 69
function keys 30
function keys. 102
Functions. 39

G

goto 53
graphics dump 87
Greater than 36
Greater than or equal to 36
greaterthan 79, 83

H

hard disc 4
header 72
headers 72
Headers and Footers. 72
hex 59
hexadecimal 60
Hidden Text 88
hide 88
hourglass 68
How is it Done. 65

I

icon bar 10, 20, 22, 91
icon bar menu 11, 47, 101
icon bar menu. 50
icon bar. 17, 76
icon. 18
ield names 76
if 35, 37, 105-106
In 9
include 73
index 80, 84
Index button 84
indexed records 80
Indexing. 80
info 45
Info. 17
Information 27
Initialising the Printer. 89
input box 13
input focus 30
insert 28-29, 63, 69
Installation 3
int 47
int function 71
Integer Conversion Operators 60
Integer Expressions and Operators 59
integer expressions. 59
integer variable 40, 47
IP 42
ISO Latin 1 89

J

Journal 11, 18, 20, 68
Journal window 25, 100
justification operator 72

K

key 30, 32
Key Definitions. 29

key strokes 13
keyboard keys 30

L

leaf name 58
LED 14
left 31
leftmargin 73
Less than 36
Less than or equal to 36
lessthan 79, 83
line 36
literal string 28
literal strings 28
Loading Files. 6
logic action 79
logical action. 83
logical expression 35
Logical Expressions. 35

M

macro 38, 55, 71, 73, 90, 104
Macro Commands. 38
macros 70
margin line 72
Margins 65, 72-73
mark 49
marked area, 51
marker 49, 52-53
marker names 51
Marking Text. 49
memory 27
Memory Text Transfer. 9
menu 91-93
menu definition 92
Menu Definitions. 91
Message Severity Levels. 45
Messages 20, 25, 45-47
mousebuffer 36
mousewindow 30-32
multi-tasking 9
Multiplication 59

N

Nesting Dynamic Items. 73
newline 63, 75
Non-Printing Characters. 62
Not equal to 36
not found 38
notequal 83

O

Obey 8
Object 99
object code 100
Object directory 8
object file 100
offer 93
open 29
Opening a Data Base File. 76
operator 78, 83
operators 59
or 36, 79, 83
Other Icons. 15
Other Simple Commands. 33
Other Uses for Menus. 93

P

page header 72
page numbering 65
pagewidth 62
parenthesis 72
Passing Parameters. 41
PASSWORD 5
paste 52
pattern matching 55
pause 88
percentage 60
PFB1 71
PIN 89
PoScript 8
POUND 90
poundwidth 60
PremCmd 7
PremData 8, 76
PremDoc 7, 18, 66, 69, 101

Premier icon 76
Premier Reference Manual. 30, 33
Premier System 1
Print 9, 87
print document 87
printed output 19
printer 10, 65
printer code 9
printer code strings 71
printer control 57
printer driver 10, 87
Printing Documents. 9
Printing Pound Signs. 89
Program Bugs. 48
prompt 104
Push Buttons. 14

Q

Quit 20, 102

R

RAM 3
Re-Open 18, 50
read 80
Reading Successive Records. 77
records 75
red font, 74
relational data base 104
ReOpen 11
repeat 36-38, 106
Repetitive Commands. 36
replace 55, 106
replacement string 56
return 63
rewind 81, 85
rightmargin 73
RISC OS 17-18, 21, 92, 102
roman number 60
roman numerals 61

S

save 10, 57, 93
Saving and Restoring Levels. 47
screen fonts 65
screen menus. 91
search 54-55
search direction 55
Searching Backwards. 55
Searching for Records. 77
security 17
Security and Validation. 4
seek 77-78, 80, 85, 105
Seek button 85
select 28
selected buffer. 28
Selecting Different Markers. 53
send 63, 70, 88, 90
serious 45
Setting Filters 83
show 22-23, 41, 53
Simple Items. 66
Simple Text Processing. 106
Single Sheet Printers. 88
Software 2
sorted order, 80
SP 42
Special Field Variables. 81
spool 57
Spooled 58
Spooling. 57
spreadsheet slots 65
Star Commands. 24
start 106
stddate 23, 95
StdKeyDefs 1
str 32, 55
str function 72, 74
string constants 32
string constants, 31
String Expressions and Operators. 61
string variables 77
sub-menu 93
Sub-Menus. 93

Subtration 59
syntax errors 100
system 46
system variable 37
system variables 32, 95
System Variables. 32
System Windows. 6

T

tab 63
tab settings 65
Text 8
text buffer, 49
text file 27, 87
Text Input Boxes. 13
text mode 87
textual data 83
The Action Buttons. 84
The ClipBoard Buffer. 50
The Command Store. 23
The Config File. 107
The Cursor Marker. 50
The Data Base Dialogue Box. 82
The Date Format String. 95
The Filer Window. 58
The Find and Search Commands. 54
The Finder Dialogue Window. 55
The Journal Window. 24
The Main File Types. 7
The Menu Command. 33
The Paste Command. 52
The Premier User's Guide. 3
The Save Command 57
The Special Format Characters. 96
The Time and Year. 95
tick 92
time 23, 74, 95
true 35

U

unique names, 17
unsaved documents 20
until 36, 106
Update 85,

V

variable length records, 82
variables 31
Variables and Constants. 31
VAT 65
VERBOSE 20, 45-46, 107
verbose messages 46

W

warning 45
while 37, 104
window 29, 31
window cursor 53
window name. 18
write 77

Y

year 23, 95